ARPA ORDER NO. 2223

12

B.S.

ISI/RR 77-61
September 1977

ADA046273

Stephen D. Crocker

State Deltas: A Formalism for Representing Segments of Computation

D D C
RECEIVED
NOV 9 1977
F

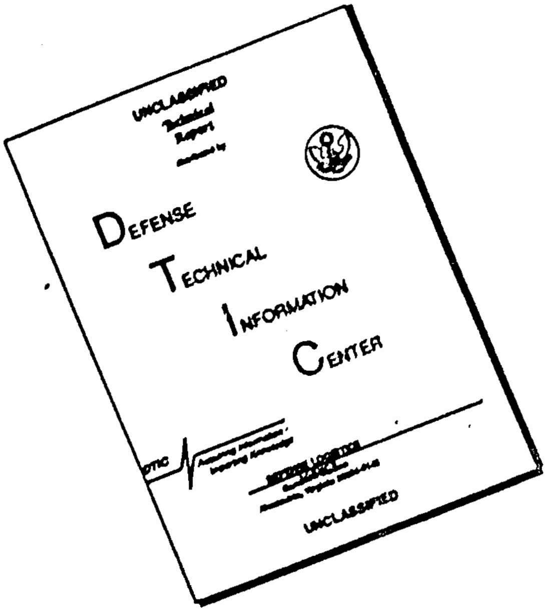
AD No.
DDC FILE COPY

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA ISI
4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Stephen D. Crocker
State Deltas: A Formalism for Representing Segments of Computation
ISI/RR 77-61

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

USC / INFORMATION SCIENCES INSTITUTE
4676 Admiralty Way   Marina del Rey   California 90291

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

(14) 1. REPORT NUMBER
ISI/RR-77-61

2. GOVT ACCESSION NO.

(9) 3. RECIPIENT'S CATALOG NUMBER

(6) 4. TITLE (and Subtitle)
State Deltas: A Formalism for Representing Segments of ~~a~~ Computation,

5. TYPE OF REPORT & PERIOD COVERED
Research / rept.,

6. PERFORMING ORG. REPORT NUMBER

(10) 7. AUTHOR(s)
Stephen D. Crocker

(15) 8. CONTRACT OR GRANT NUMBER(s)
DAHC 15-72-C-0308,

9. PERFORMING ORGANIZATION NAME AND ADDRESS
USC/Information Sciences Institute ✓
4676 Admiralty Way
Marina del Rey, CA 90291

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
✓✓ARPA Order ~~#~~2223
Program Code 3D30 & 3F10

11. CONTROLLING OFFICE NAME AND ADDRESS
Defense Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209

(11) 12. REPORT DATE
Oct[redacted] 1977

13. NUMBER OF PAGES
127

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
(12) 128p.

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document approved for public release and sale; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

D D C
RECEIVED
NOV 9 1977
RECEIVED
F.

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
ARPANET IMP, correctness, first-order predicate, program verification, proof system, state delta

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(OVER)

1B

407 952

20.

A state delta is a first-order predicate taking a precondition, a postcondition, a modification list and an environment list. State deltas are to be used in a proof system in which predicates describing the machine's current state are cross-referenced by the locations containing values used in each. Reasoning about a machine's forward progress may be carried out by successive application of state deltas. Application is legal if the precondition is true and if none of the locations in the environment list has been changed since the state delta was entered into the system. Afterward, all predicates cross-referenced to any of the locations in the modification list are deleted, and those in the postcondition list are added to form a new current state. If some locations hold arrays or lists, the cross-referencing part of the system may provide for attaching predicates to location components. If names may be given to individual location components, different predicates may depend upon the same values under different names. To keep track of all ~~the~~ overlap conditions possible among a set of named locations, a graph structure with two node types is used to show all ~~the~~ possible overlaps. A small proof system has been built and used to prove the correctness of a slice of the code in the ARPANET IMPs.

ARPA ORDER NO. 2223

ISI/RR 77-61
September 1977

Stephen D. Crocker

State Deltas: A Formalism for Representing Segments of Computation

ACCESSION for
NTIS White Section ☑
DDC Buff Section □
UNANNOUNCED □
JUSTIFICATION

BY
DISTRIBUTION/AVAILABILITY CODES
Dist. SPECIAL
A

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308, ARPA ORDER NO. 2223, PROGRAM CODE NO. 3D30 AND 3P10.

VIEWS AND CONCLUSIONS CONTAINED IN THIS STUDY ARE THE AUTHOR'S AND SHOULD NOT BE INTERPRETED AS REPRESENTING THE OFFICIAL OPINION OR POLICY OF ARPA, THE U.S. GOVERNMENT OR ANY OTHER PERSON OR AGENCY CONNECTED WITH THEM.

THIS DOCUMENT APPROVED FOR PUBLIC RELEASE AND SALE; DISTRIBUTION IS UNLIMITED.

blank

Contents

Acknowledgments

The research reported here was carried out over a three-year period while I have been a staff member of the Information Sciences Institute of the University of Southern California. Interactions with colleagues at UCLA, ISI and elsewhere have been invaluable in developing the ideas presented here. Charlie Hayden and Dono van-Mierop helped implement the ISPS translation program and the proofchecker; this project would surely have been less successful without their help.

Discussions with Susan Gerhart, Ralph London, Mac McKinley, Dave Musser, Ron Tugender and Dave Wile of ISI, with Bill Overman of UCLA, with Bill Carter and Bill Joyner of IBM, with James H. Morris Jr. and Ben Wegbreit of Xerox PARC, with Zohar Manna of Stanford University, with Richard Waldinger of SRI, Rod Burstall of Edinburgh University and with Vint Cerf of ARPA all contributed to a greater understanding of the problems involved in specification and proof of computer programs.

In addition to benefits gained from personal discussions, two papers stood out as containing essential insights into the problems of representing computational activity. John McCarthy's "Situations, Actions and Causal Laws"[1] introduced the notion of *fluents* which are implicit functions of the current state of the world. The formulation of state deltas is closely related to McCarthy's fluents, although there are specific differences noted in the text.

The other paper of major importance is Rod Burstall's "Some Techniques for Proving the Correctness of Programs which Alter Data Structures."[2] The fundamental idea is simply a bookkeeping system for partitions, and this is applied to the problem of keeping track of what space is covered by which data structures. This paper crystallized some ideas I had been playing with and became the basis for the design of the place system for handling overlap relationships. I'm particularly indebted to James H. Morris Jr. for directing me to that paper.

[1] John McCarthy, "Situations, Actions and Causal Laws," *Semantic Information Processing*, edited by Marvin Minsky, MIT Press, Cambridge, Massachusetts, 1968, pp. 410-417.

[2] Rod M. Burstall, "Some Techniques for Proving Correctness of Programs which Alter Data Structures," *Machine Intelligence 7*, Edinburgh University Press, Edinburgh, Scotland, 1972, pp. 23-50.

The original goal was to formally specify the operation of the IMP program and prove that its code was correct with respect to the specification. Although this goal was set aside in favor of re-examination of the basic formalisms, the early part of the work was devoted to an intensive study of the IMP code and the architecture of the Honeywell 316. During this period, a number of people at the IMP project at Bolt, Beranek and Newman in Cambridge, Massachusetts, including Ben Barker, Herb Brown, Steve Butterfield, Bernie Cosell, Bill Crowther, Frank Heart, Joel Levin, Alex McKenzie, Paul Santos and Dave Walden, provided extensive assistance and consultation.

As part of the early effort to formalize the description of the Honeywell 316, I experimented with Bell and Newell's ISP notation. Mario Barbacci, Gary Barnes, Rick Cattell and Dan Siewiorek of Carnegie-Mellon University developed a revision of the notation called ISPS and wrote a number of programs to process ISPS descriptions. The key tool is a parser which accepts ISPS descriptions in source form and outputs a fully parenthesized parse tree in a text file for further processing. With their assistance, I was able to use their programs over the ARPANET and experiment with translation of ISPS descriptions into both executable programs and sets of state deltas for use in proofs.

The tools available to use during the course of a research project always have a major impact on the outcome. I was fortunate to benefit from very powerful tools and from help from experts. By far the most important tool is Interlisp. Years of refinement have produced a system far more sophisticated and useful than the original implementations of LISP 1.5. The file package, record package, read macros, CLISP extensions, pattern matcher and dwimifier all played major roles in the design and implementation of the programs. As powerful as these facilities are, however, they would hardly be useful without supporting documentation. The Interlisp manual[3] is a masterpiece of documentation and surely represents one of the most successful products to come out of the computer science research community. Beyond the production of their excellent system, Alice Hartley, Daryle Lewis, Larry Masinter and Warren Teitelman also exhibited great patience in answering questions and responding to problems. At ISI, Marty Yonke shepherded our local version of Interlisp and protected Warren and the others from the more naive questions.

Document preparation plays an important role in any project, perhaps even more so when a dissertation is involved. This document was prepared using PUB, a rather elaborate but very temperamental system. Ray Bates, Norton Greenfeld and Marty

[3] Warren Teitelman, *Interlisp Reference Manual*, Xerox Palo Alto Research Center, Palo Alto, California, 1975.

Yonke provided extensive assistance in building the macros necessary to drive PUB and in guiding the secretaries and me through the PUB maze. Actual printing of the documents were carried out on the XGP, also a temperamental device. Very little trouble was encountered with the XGP, however, because Pete Alfvin mastered the XGP software completely. Pam Kaine and Rennie Simpson typed this document and wrestled with PUB to make it look right. Their patience and perserverance cannot adequately be rewarded.

This work was carried out under the guidance of UCLA faculty members Ken Colby, Donald Kalish, Leonard Kleinrock, David Martin, Gerald Popek and Gerald Estrin. As a committee and as individuals, these people provided exactly the kind of probing and feedback needed at critical points in a research project. Professor Estrin served as chairman of the committee, but his role went well beyond the mere formalities of chairing a dissertation committee. He has been a friend, a counselor, a colleague and a teacher; I could not have found a more supportive, patient, tenacious or insightful adviser.

Financial support for the work reported here came from the Advanced Research Projects Agency and the Rome Air Development Center, both through contract DAHC-72-C-0308. The financial support and the personal support of Bill Carlson, Dick Nelson and Don Roberts are gladly acknowledged.

Norton Greenfeld, Chloe Holg, Bob Metcalfe, Vint Cerf and my irrepressible wife Beth have provided moral support and encouragement during this period, and my pleasure in completing this work comes partly from their participation.

Finally, despite the help and encouragement of all of these people, the views expressed here are purely my own and neither the people nor agencies named above may be held responsible for errors or misstatements.

Summary

A *state delta* is a form for representing segments of computation. A state delta is a first-order predicate which takes a precondition, a postcondition, a modification list and an environment list. The semantics of a state delta are "if the machine is in a state which satisfies the precondition, it will eventually reach a state which satisfies the postcondition, and it will do so without modifying any locations except (possibly) those listed in the modification list."

It is intended that state deltas be used in a proof system in which predicates which describe the current state of a machine are cross-referenced according to the locations containing values used in each predicate. In such a system, reasoning about the forward progress of a machine may be carried out by successive application of state deltas. Application of a state delta is legal if the precondition is true of the current state and if none of the locations in the environment list has been changed since the state delta was entered into the system. After these checks have been made, application consists of deleting all predicates cross-referenced to any of the locations in the modification list and adding the predicates in the postcondition to form a new current state.

If some of the locations in the machine being modeled hold arrays or lists, the cross-referencing part of the proof system may provide for predicates to be attached to components of the locations. If the proof system further allows for names to be given to individual components of locations, it is entirely possible for different predicates to depend upon the same values under different names. Under these circumstances, it is necessary to keep track of all of the overlap conditions possible among a set of named locations. In most cases of interest, a graph structure with two types of nodes can be employed to represent all of the possible overlaps, and this graph can be searched quite efficiently whenever the current state is to be modified.

A small proof system has been built along these lines and has been used to prove the correctness of a slice of the code in the IMPs in the ARPANET. This code allocates a buffer off the free bufferlist and uses indirect addressing and other pointer techniques to accomplish its task. Representing the allocation of storage and following the trail of the pointers fully illustrates the utility of the graph system for representing storage relationships.

1. Introduction

I have been interested in applying formal verification techniques to real programs. What's a "real" program? One whose reason for being written is to accomplish some computational task and not solely to serve as an example for verification research.

I chose the code for the IMPs in the ARPANET as a real program to study.[4] For the present discussion, it is sufficient to know that the code consists of about 10,000 instructions (including load-time constants), runs on a Honeywell 316, which is a rather standard minicomputer, and is written in the assembly language for the machine. Its purpose is to ship messages around the network from source "hosts" to destination hosts.

When I began looking into the IMP code, I hoped that existing theory and techniques would be sufficient. My task would be only to extract the relevant details from the IMP code and submit them to an existing verification system. I was prepared, of course, to supply all of the assertions that might be required, define the concepts and terms specific to the IMP code, and write some sort of preprocessor to transform the machine language (or its assembly level representation) into some more pleasant form consumable by existing verification systems.

It didn't take long to find out this approach wouldn't work.

Two classes of problems emerged. First, existing verification systems are still in an early stage of development. Verifying a program of any size would require both extension of the existing system to handle primitive operations such as anding and oring of bitstrings and extensive interactive direction to drive the system. Were these the only problems, the right course would have been to help extend an existing system.

The problems in the second class are more fundamental. I could not find a way

[4] The ARPANET is described by Larry Roberts and Barry Wessler in "Computer Networks to Achieve Resource Sharing", *Proceedings of the AFIPS Spring Joint Computer Conference*, Vol. 36, American Federation of Information Processing Societies, Montvale, New Jersey, 1970, pp. 543-549. For a description of the IMP program, see "The Interface Message Processor for the ARPA Computer Network", by Frank Heart, *et al.*, pp. 551-567 in the same volume.

to map the following facts and practices pertaining to the IMP code into the formalisms accepted by existing verification systems.

> Indirect addressing, computed branches and program modification are used extensively.
>
> Pointers are used extensively to manage space and eliminate expensive copying of data from one location to another.
>
> All of the code and all of the data share a common residence -- the memory of the machine. Any proof of correctness of the system needs to include a proof that data and programs assumed to be separate from each other are indeed disjoint and do not clash.
>
> The code consists of a number of routines driven by interrupts and operating concurrently. Some of the interactions are timing-dependent; slowing down or speeding up the processing time for some of the routines could cause the system to fail.

Why are these aspects of the IMP code unacceptable to existing verification systems? Current efforts to build program verification systems are based on Floyd's approach.[5] In these systems, a program is represented in a flowchart form and augmented with assertions which relate current values of the various program variables. The assertions are then combined with the program text to produce a set of lemmas -- called *verification conditions* -- to be proven. Simplification and theorem-proving programs are then used to prove each of the lemmas. When all of the lemmas have been proven, the program is guaranteed to be consistent with the assertions embedded in the program; in particular, if one of the assertions is attached to an entry arc and one to an exit arc, then this pair of assertions forms the input-output assertions for the program. Systems built along these lines have a number of limitations.

> Generation of the verification conditions is completely automatic and thus depends upon a purely syntactic analysis of the program. For programs

[5] The theory is outlined in "Assigning Meanings to Programs," by Robert W. Floyd, printed in *Mathematical Aspects of Computer Science*, Vol. XIX, Proceedings of Symposia in Applied Mathematics, American Mathematical Society, Providence, Rhode Island, 1967, pp. 19-32. A typical system based on this theory has been implemented under the direction of Ralph London. "An Interactive Program Verification System," by Donald I. Good, Ralph London and W. W. Bledsoe, published in *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1, pp. 59-67, March 1975, describes the system.

written in restricted high-level languages, such syntactic analysis may be possible. However, for programs written at the assembly level or in rich languages which permit program modification and computed branches, determination of the paths must be combined with the proof process itself. Another facet of the same problem is that the programmer may want to analyze his program in terms of execution sequences which do not correspond to successive values of the program counter. Table-driven or interpretive systems may look like reasonably simple repetitive loops if only the apparent flow of control is followed, whereas the actual complexity of the program may be buried in the "data".

No distinction is made between the value stored in a program variable and the name of the program variable itself. Without this distinction it is unclear how to represent the state of affairs for programs using pointers, indirect addressing and related practices.

Arrays are usually treated as closely as possible as simple variables so that assignment to a component of an array is considered to have changed the value of the entire array. As a consequence, every fact known about the array must be rederived after an assignment to any part of it. This is particularly disastrous for verification of machine language programs; all of memory is one array!

There are simply no provisions at all for treating timing-dependent code.

In addition to these fundamental limitations, there are also some engineering considerations that materially limit the utility of existing systems.

Termination is one of the most common properties requiring verification. Unfortunately, Floyd's theory treats termination quite separately from attainment of correct values. A completely separate set of assertions and a separate proof are required, approximately doubling the work. This cost has not yet been passed on to the user, however, because the implementers of verification systems have not yet built the additional machinery to handle termination.

The separation of the verification process into two distinct stages, generation of verification conditions and proof of the verification conditions, means that much of the structure of the program is inaccessible during the proof process. In my view, this is an unnatural separation which will tend to make verification unnecessarily difficult in practice.

Current practices require that the programmer write down all facts relating to the current state in every assertion. Since an assertion is required for every loop, the programmer will frequently need to copy facts which are irrelevant to the loop but which must be brought forward across the loop. For example, if one section of the program sets the value of FOO to be zero, and the next section uses a loop to clear an array but does not change FOO in the process, the assertion attached to the loop for clearing the array must nevertheless include a clause stating that the value of FOO is zero.

One possible response is to suggest that the IMP code is a poor choice for verification and that only programs which satisfy a particular set of constraints should be considered for verification. This point of view is usually espoused under the "structured programming" label or, more recently, under the "quality software" label.

In general, I can agree that structuring techniques should be used whenever possible and I am a strong proponent of higher-level languages and strong conventions, *provided the languages and conventions do not interfere with the accomplishment of the primary task*. In many cases, however, languages for programming an important application on a particular machine either do not exist or else impose intolerable time or space limitation. In such cases, assembly language or other loose forms of code generation are essential.

More to the point, perhaps, is a strong feeling on my part that the essentials of program verification need not be tied to any particular language. In my own experience as a programmer, I find that there is only a little variation in the analyses of (say) LISP programs versus machine language programs.

Finally, from a common sense viewpoint, the IMP code is not very complex. Most competent programmers would find it possible to learn the code and relate observed behavior of the IMP to the structure and content of the code.

As a consequence, I prefer to take programs like the IMP program as fixed points on the landscape and ask whether there are techniques possible for analyzing them and proving their properties.

This line of inquiry has yielded the following results.

A new form for representing a segment of computation has been invented. The new form is a *state delta* (SD). The essential components of a SD are a *precondition*, a *postcondition* and a *modification list*. The pre- and postconditions are predicates on the machine's state vector and the modification list is a set of names of components of the state vector. The semantics of a state delta is "if the machine is in a state which satisfies the precondition, it will eventually reach a state which satisfies the postcondition, *and in the course of the intervening computation, only the places listed in the modification list may be changed.*" The modification list is the mechanism which relieves the pre- and postcondition of the burden of copying static information forward through a proof.

State deltas are first-order predicates in their own right, and a proof theory has been developed which provides the logical machinery for using state deltas in proofs and proving new state deltas. The proof theory is a direct extension of the theory in Kalish and Montague's treatment of first-order predicate calculus with identity and definite description[6] and contains two new inference rules. One of these rules provides the basis for forward reasoning through the sequential parts of a program. The other provides for combination of state deltas which cover alternative paths. No new inference rules are required for loops; the usual forms of mathematical induction are completely adequate.

One of the intended uses for state deltas and the new proof machinery is to provide rigorous proofs of correctness of code which involves indirect addressing and list structures. This requirement led to an understanding of the role of placenames, i.e., names of components of the state vector. In most theories used in program verification, these names are fixed and represent disjoint components. Part of the theory developed here provides machinery for referring to overlapping components of the state vector and for using proof variables as well as constants to refer to parts of the state vector. In essence, all possible overlap relationships among the placenames in use must be known. A compact form for representing this information and an extremely efficient method for searching the representation to find the most highly intersecting set of places has been developed.

Based on the proof theory developed, a trial verification system has been developed and tested. The system is a blend between a proofchecker and a symbolic execution system. State deltas are used to progress forward through a computation symbolically and a complete record is kept of the effects. One of the design goals for this implementation was that the cost of making a single step should be nearly independent of the size of the state description at any point. This goal was only partly realized, but the difficulties became clear and provide direction for future implementations.

[6] Donald Kalish and Richard Montague, *LOGIC: Techniques of Formal Reasoning*, Harcourt, Brace and World, Inc., New York, 1964.

As part of the experimentation with the verification system, the operation of the 316 was coded into a set of state deltas. This led to an exploration of other forms of description of machine behavior, and an experiment was carried out to translate machine descriptions in Bell and Newell's ISP notation into state deltas. This translation was successful, but an excessive number of state deltas were generated. I developed an alternative scheme for translating ISP descriptions into far fewer state deltas, but have not implemented this new scheme. One byproduct of the experimentation with ISP has been a set of interactions with other researchers using machine descriptions for simulation, code generation, hardware design and other applications. Out of these interactions there is emerging sufficient experience to design a machine description language which is rigorously defined and suitable for a wide spectrum of applications.

Finally, the system was used to prove the correctness of a tiny section of the IMP code. This section of the IMP code allocates a buffer from a free storage list. The proof of correctness of this section of the IMP code required specification of the list structures used in the IMP code and provides a blueprint for verifying related implementation of other list processing code. The proof itself is quite long and reflects the primitive nature of the verification system. As may be expected, however, the tedium of preparing a long proof of very small steps has provided substantial guidance for future improvement of the verification system.

In addition to the limitations mentioned above, the present work makes essentially no contribution toward an understanding of concurrency and timing. The concept of keeping track of what may be changed between two points in time seems to be necessary, but a much stronger formalism will need to be created to represent the interactions among multiple processors.

These results are elaborated in the succeeding chapters. Chapter two contains the detailed formulation of state descriptions and state deltas. Chapter three describes the structure of a small proofchecker which uses and proves state deltas. Chapter four is devoted to a description of the IMP and and a description and informal proof of correctness of the buffer allocation routine, named GFREE. A complete formal proof of correctness of this same code has been verified by the verification system and it is displayed and annotated in chapter five. Chapter six contains reflections on the current work and directions for the future.

All of the programs developed during the course of this research were written in Interlisp. This choice was intentional, for it was clear at the outset that only the availability of such a powerful system would enable one person to experiment with a theory by implementing and re-implementing various ideas. Interlisp not only supported all of the programming I needed to do, but it also supported the definition of language for state descriptions and state deltas and provided a powerful pattern matching facility for use in a command language for the verification system. These uses of Interlisp fall somewhat outside the design intentions of the Interlisp architects and the fit was not quite perfect; some of the syntax may seem a little awkward For a production version of the verification system, it is almost certain that all of the interfaces would have to be redesigned.

2. The Formalism

The pattern for proving facts about a program is to represent each of its steps as a state delta and then prove facts about sequences of steps. The new facts will also be represented as SDs and thus may be used in further proofs.

We will come to the precise formulation of SDs shortly, but we know that they contain a modification list and two partial state descriptions, viz. the precondition and the postcondition. Given a set of known SDs, we will attempt prove a new SD in the following manner.

1. Write down the precondition for the SD to be proven. This constitutes the initial "current state".

2. Select a known SD whose precondition is true for the current state and apply it. After the SD is applied, there will be a new current state.

3. If the new current state meets the requirements of the postcondition in the SD being proven, the proof is finished. If not, step two is repeated until a state is reached that is satisfactory.

"Application" of a SD to the current state has two subparts. First, clauses in the current state that depend upon the contents of one or more places being modified must be removed from the current state. After this has been done, the postcondition of the SD being applied is added to the state and the total result is the new current state.

2.1 Descriptions

A computer has a set of *places* (sometimes called *locations*) which hold *values*. The collection of values stored in the places at a given time is the *state* of the machine at that time.

Places are either *simple*, *structured* or *invented*. Simple places hold non-negative integers in the range 0 through 2^{n-1}, where n is the *length* of the place. Simple places are used to model single registers or flip-flops.

Structured places hold lists. Structured places are used to model the memory array and other places which hold more than one element. As we will see later, structured places will also be used to model subsections of memory, including non-contiguous subsections.

Invented places are used to map control into the state description. Our only use of invented places in the present work is to model a fictitious microprogram counter and a set of places to hold "return addresses" for the microprogram. The values stored in invented places are just labels with no intrinsic structure. The only operations available for labels are movement from one place to another and comparison for equality.

I considered formulating simple places as holding bitstrings and defining the various operators accordingly. Selection of a field from a word is a typical operation performed on values in simple places and has a very simple definition in terms of bitstrings. The major drawback of using bitstrings, however, is that there has to be an interface to the integers at some point, and it becomes tedious if carried out at the bit level. For example, after selecting a single bit from a bitstring, the value is still a bitstring (of length 1) and is not officially comparable to an integer 0 or 1. Just the matter of writing down constants becomes a chore: either a constant is an integer and must be explicitly converted to a bitstring of some length, or it is initially a bitstring and the length must be specified along with its value.

Using integers as the values for simple places turned out to be easier than I had first guessed. Selection of fields and other "bitstring-oriented" operations can be characterized in terms of integer division (remainder and quotient) and the interface between selection of elements from an array and selection of "bits" from an integer can be formulated relatively cleanly.

Our concern is almost always with a set of related states instead of just a particular state. To describe just the set of states of interest, we use a *state description*. A state description is just a list of clauses in the first-order predicate calculus. The list is understood to be a conjunction. In addition to the logical connectives, quantifiers and equality, a number of operators and predicates are predefined. The user may also define his own operators and predicates. The predefined operators are introduced below. For each of these operators, there is an infix form and an prefix form. The verification system accepts either form for input and converts all inputs to prefix for processing. When clauses are output, they are converted to infix form.

2.1.1 Contents-of

I use a *contents-of* operator to refer to the value stored in a place in a given state. Its external syntax is a unary prefix period and its internal form is DOT.

The use of a contents-of operator provides for a distinction between the name of a place and its contents. We will need this distinction in analyzing indirect addressing computations and structures involving pointers.

Using the contents-of operator, we can write an example of a simple state description:

```
.PC=5 and .A=0.
```

This state description refers to any state which has 5 stored in PC and 0 in A . The values in all the other places are unconstrained.

2.1.2 Selection

Many of the places will be considered to hold arrays or sequences. In order to refer to a particular element, a *selection* operator is provided. The external syntax for the selection operator is ∘ and its internal form is SEL. MEM∘5 represents the sixth element of MEM.[7] (All arrays and sequences are indexed from zero.)

MEM∘5 is the name of a place. Its contents are .MEM∘5. This notation is potentially ambiguous, for it is not clear whether the selection operator or the contents-of operator has higher precedence. If the contents-of operator has higher precedence, then .MEM∘5 means (.MEM)∘5. This means that the value of the whole array is first obtained and then element 5 is extracted. This interpretation means that the selection operator would be operating on a *value* instead of a place.

In contrast, if the selection operator has higher precedence, .MEM∘5 will be

[7] The choice of notation is heavily influenced by the availability of CLISP which translates automatically from internal to external syntax and back again. The convenience of this facility has outweighed the nuisance of using a non-standard notation. For a production system, I expect that a different external syntax would be developed.

interpreted as .(MEM∘5), meaning the contents of the place designated as the sixth component place of MEM.

Either of the these interpretations should lead to the same value, but we will see that it is desirable to minimize the size of the places that appear under the contents-of operator. Accordingly, I have chosen the latter interpretation for .MEM∘5. Of course, the former interpretation is still available if extra parentheses are supplied.

Selection is also defined for integers; the result is equal to the corresponding bit in the binary representation of the integer, i.e., x∘0 = 0 if x is even, etc.

2.1.3 Indexof

Given a placename like FREE, it is often desirable to find its address in memory. If FREE=MEM∘i for some i, we'd like a way to refer to i. The *indexof* operation is provided for this purpose.. Its external syntax is an infix /; its internal syntax is indexof. If FREE=MEM∘i, then i=FREE/MEM. The second name must be a structured place and the first name must be one of its elements.

2.1.4 Segthru and Segfrom

These operators select subsequences of structured places, structured values or integers. (segthru X N) extracts elements 0 through N of X. If X is an integer, (segthru X N) takes X modulo 2^{N+1}. The external syntax for segthru is ";".

(segfrom X N) extracts elements from N through the end. If X is an integer, (segfrom X N), is the integer quotient of X divided by 2^N. The external syntax of segfrom is ",".

Typical use of these operators is to extract a bit field from a word. .M;13,10 extracts four bits, bits 10, 11, 12 and 13, from registers M and returns an integer in the range 0 through 15. Reversing the operators provides a way of stating the first element and the number of elements: .M;13,10=.M,10;3. Note that X,0=X. When working with integers, X∘i = X;i,i. However, for structured places and values, X;i,i has the same number of dimensions as X with the outer dimension equal to 1, while X∘i has one less dimension than X. Consequently, X∘i = X;i,i∘0 = X,i∘0.

Following the policy established for SEL, segfrom and segthru have lower precedence than OUT. When more than one occurs, they are performed left to right.

2.2 Overlap among places

Usually, each place has a single name and is isolated from all other places. Changing the value stored in one place doesn't affect the value stored in another place.

Arrays, indirect addresses and list structures all require a different point of view. In one form or another, these mechansims each involve dynamically changing placenames. As a consequence, we may not know whether two names refer to the same or different places.

Our general plan for following programs is to step through them symbolically. Whenever an assignment is made to a place, its old value will be discarded. Values in other places remain undisturbed until assignments are made into those places.

This plan clearly requires that we know which places are disjoint from each other whenever an assignment is made. How do we resolve these conflicting requirements?

First, we adopt the conservative rule that unless we know that two places are disjoint, we must assume that they might overlap. Second, we provide a fast and reasonably flexible mechanism for recording and accessing the overlap relationships.

Since our default rule is that places overlap unless we know definitely that they do not, we need some means of saying that two places do not overlap. We could adopt a predicate, say Disjointp(x y), which asserts that x and y are disjoint, and then we could make up axioms for deriving disjointness from other properties. If we tried to do so, we would encounter a major difficulty in working with a large number of places. To assert that three places are each disjoint from each other takes three statements, four places requires six statements, five takes ten statements, etc. In practice we need to assert that perhaps several hundred places are each disjoint from each other; several thousand individual occurrences of Disjointp would required. This difficulty could be remedied by expanding the predicate to take an indefinite number of arguments. The semantics would be that each of the arguments is pairwise disjoint from each of the others.

In addition to specifying which places are disjoint from which other places, there is frequent need to specify that one or more places are wholly contained within another place. Both of these concepts -- disjointness and subterritory -- are common, and I have chosen to combine them into a single predicate, Covering. (Covering <A B_1 B_2 ... B_n>) states that B_1 through B_n are disjoint from each other and that they are all contained within A.

It is not sufficient to have these relationships scattered about in the state description. When a modification is made to one of the places, it is essential to know what other places may overlap with the modified place. To speed up the search among these relationships, a separate data structure is maintained which duplicates the information contained in the Covering predicates and provides immediate access to all of the interactions among them. This data structure is called the *place graph* and is explained in detail in the next chapter. The most important point about the place graph is that every place that is referenced within a proof is expected to be listed in the place graph. Places listed in the place graph are said to be *registered*. Because the place graph expects to know about all places, it assumes that the Covering relationships it knows about are definitive and that all overlaps which are not explicitly barred might actually exist.

Although the default assumption is that two places overlap if it is not known that they do not, the place graph is organized so that its connections show what *does* overlap (or at least might overlap). As a consequence, the actual searches of the place graph touch only the nodes corresponding to possible overlapping places. Since most places do not overlap with most other places, the searches of the place graph tend to be independent of the size of the place graph. This is an important result and contributes significantly to the design goal of a symbolic execution system whose execution time for a single step is independent of the size of the program.[8]

The place graph is initially set to hold a single node corresponding to the place OMEGA. OMEGA represents all of the space in the machine and everything is considered to be a subplace of OMEGA. Relationships are added to the place graph by attachment to existing nodes, so the first relationship added to the place graph must be (Covering OMEGA ...).

[8] This formulation of how to handle the problem of overlapping names was influenced by Rod Burstall's elegant paper "Some Techniques for Proving Correctness of Programs which Alter Data Structures," in *Machine Intelligence 7*, Edinburgh University Press, Edinburgh, Scotland, 1972, pp. 23-50. I am indebted to James H. Morris Jr. for pointing out this paper during a discussion on the subject of how to represent list structures.

2.3 Computation

State descriptions provide a characterization of a machine at a single point in time. The next step is to describe the computational process as the machine proceeds from one state to another. The basic requirement is for some way to state "if the machine is in a state characterized by P, it will eventually reach a state characterized by Q", where P and Q are state descriptions. For brevity we can write "P leads to Q".

The general plan is represent the hardware as a set of statements in this form and then combine these statements together to cover long sequences of computation. Thus, if we have "P leads to Q" and "Q leads to R", we expect to be able to write "P leads to R". However, this notation by itself suffers from a problem mentioned in the introduction: everything relevant to later computations needs to be included in each of the state descriptions between the first time it becomes true and the last time it is used.

To avoid this burden, an explicit list of places which are modified is included in the description of the the computation. The semantics of a computation are now "if the machine is in a state characterized by P, it will eventually get to a state characterized by Q, *and it will do so without modifying the contents of any place except (possibly) those listed in* M." All of this is summarized by `(SD(pre:P) (mod:M) (env:) (post:Q) (vars:))`.

The vars clause binds variables that appear in P, Q and M. It is simply a convenient alternative to using a universal quantifier.

```
∀x(SD (pre: P(x)) (mod: M(x)) (env:) (post: Q(x)) (vars:))
```

is identical to

```
(SD (pre: P(x)) (mod: M(x)) (env:) (post: Q(x)) (vars: x)).
```

Statements of this form are called *state deltas* and form the basis for reasoning about the computational process. We can still combine two state deltas to form a third by extending our previous notion a bit: if P leads to Q modifying only places in M and Q leads to R, modifying only places in N, then P leads to R modifying only places in M∪N.

The env clause is a list of places; it is used to abbreviate the precondition. In order to characterize the operation of a subroutine, the precondition would need to include a listing of the code as well as the constraints on the data structure. Listing the code in the precondition is unwieldy and inefficient. If the code is never modified, we would like to avoid both the repetition of the code in the precondition and the cost of rechecking that it hasn't changed.

When the SD is entered into the proof system, it is connected to the current machine state by its environment clause. When the SD is used, the values in the environment places must not have been changed since the SD was entered into the system. If they had been changed, the SD is no longer valid and must be discarded.

2.4 Definition of the 316

Below is an abbreviated description of the Honeywell 316 in terms of SDs. Only six of the instructions usually found on 316s are included here. This example will be used later as part of the input for the proof of GFREE. The SDs in the definition of the 316 will not have any places listed in their environment clauses, since their validity does not depend upon some part of the machine state remaining constant. We will see examples of the use of the env clause later.

The machine is assumed to have a memory MEM, a program counter PC, an A register A, and internal registers M, OP, I and UPC to hold an address, operation code, indirect addressing flag and microprogram locations, respectively. These internal registers are all considered to be subcomponents of Q, so a change to Q changes (perhaps) all of the internal registers.

This use of Q is merely a convenience; the list of M, OP, I and UPC could have been written explicitly in the mod clauses of the SDs, but it was convenient to summarize the set with a covering place.

The following list of SDs is the actual set used in the proof of GFREE described later. The place named UPC is an invented place, and it represents a microprogram counter. For this simple example, its values are represented as top, addr and action.[9]

```
(SD (pre: .UPC=top .PC=pc .MEM∘(.PC)=140040Q)
    (mod: Q PC A)
    (env:)
    (post: .UPC=top .PC=(pc+1);13 .A=0)
    (vars: pc))
```

[9] The suffix Q on numbers indicates the number is octal. This convention was chosen to conform with the available facilities in Interlisp.

```
(SD (pre: .UPC=top .PC=pc .MEM▫(.PC)=101040Q)
    (mod: Q PC)
    (env:)
    (post: .UPC=top (if .A=0
                        then .PC=(pc+1);13
                      else .PC=(pc+2);13))
    (vars: pc))

(SD (pre: .UPC=top .MEM▫(.PC);13,10~=0)
    (mod: Q)
    (env:)
    (post: .OP=.MEM▫(.PC);13,10 .UPC=addr .I=.MEM▫(.PC)▫15
           (if .MEM▫(.PC)▫9=0
               then .M=.MEM▫(.PC);8
             else .M=.MEM▫(.PC);8+(LOGAND .PC 377000Q)))
    (vars:))

(SD (pre: .UPC=addr .I=0)
    (mod: UPC)
    (env:)
    (post: .UPC=action)
    (vars:))

(SD (pre: .UPC=addr .I=1 .M=m)
    (mod: UPC I M)
    (env:)
    (post: .UPC=addr .M=.MEM▫m;13 .I=.MEM▫m▫15)
    (vars: m))

(SD (pre: .UPC=action .OP=1 .M=m)
    (mod: Q PC)
    (env:)
    (post: .UPC=top .PC=m)
    (vars: m))

(SD (pre: .UPC=action .OP=4 .M=m .PC=pc)
    (mod: Q PC MEM▫m)
    (env:)
    (post: .UPC=top .PC=(pc+1);13 .MEM▫m=.A)
    (vars: m pc))
```

```
(SD (pre: .UPC=action .OP=10 .M=m .MEM∘m=v .PC=pc)
    (mod: Q PC MEM∘m)
    (env:)
    (post: .UPC=top .MEM∘m=(v+1);15 (if .MEM∘m=0
                                          then (.PC=pc+2);13
                                        else .PC=(pc+1);13))
    (vars: m pc v))

(SD (pre: .UPC=action .OP=11 .PC=pc .A=a .M=m .MEM∘m=b)
    (mod: Q PC A MEM∘m)
    (env:)
    (post: .UPC=top .PC=(pc+1);13 .A=b .MEM∘m=a)
    (vars: pc m a b))
```

2.5 Computation of support

By *support* I mean the set of places whose contents are referenced in the expression. The truth value of a predicate depends upon the contents of the places referenced remaining the same. If any of the contents are changed, then the truth of the predicate is suspect. Our strategy will be to delete predicates from the list of known-to-be-valid predicates whenever any place in its set of support is changed. This strategy is conservative and can never lead to an inconsistency. However, it is quite possible for information to be lost. For example, the predicate .X-.X=0 is always true, no matter how the contents of X are modified. The algorithms I am using to compute support treat this predicate as if it is supported by X. As a consequence, some care by the user may be necessary in constructing his predicates.

The basic rule for computing support is just to collect the set of places that occur within a predicate under the scope of the contents-of operator (DOT). There are several exceptions, however. The following is the precise formulation used in the system.

Two functions are used to compute the support of an expression, Support and Structure. Structure analyzes expressions containing names of places which are used within the scope of a DOT. Support takes the union of the support of the subexpressions until it reaches a DOT or another special form. When reaching a DOT, Support uses the structure of the referenced place or place-expression.

The special cases are the following.

State deltas have an explicit representation of their support, declared when they are proven. In the state delta this is the environment clause.

Quantified expressions behave as if the structure of each of the bound variables is OMEGA.

The support of expressions headed by a user-defined name is dependent upon the definition of the name. When the name is defined, the user is responsible for declaring the rule to be used in computing the support of expressions headed by the name. The default rule is to take the union of the support of each of the arguments to the expression; if the default rule is correct for a particular name, then no specific rule need be stored and Support will assume the default rule. If a special rule is necessary, it must be in the form of the union of some subset of (usually all) of the arguments and a list of constants.

In all cases, when a new name is introduced, the user must prove that the rule for computing support of expressions headed by that name is correct. "Correct" means that the rule computes a list of placenames which entirely covers the places holding values used in the definition of the predicate. For predicates which are not defined recursively, this simply means that the list of places computed by the rule must cover the list of places computed by applying Support to the definition of the predicate.

For predicates defined recursively, the same criterion is used, except that the proposed rule for computing support is used in the appearances of the new predicate in the definition.

The following example illustrates computation of support for a new predicate. (PacketBufferListp x y) is defined to be

```
(Subset x Bufferspace) and
y≠0 and
(y is less than 377770) and
(if (LENGTH x)=0
    then y ZERO/MEM
  else (PacketBufferp x) and
        y x∘0∘0/MEM and
        (PacketBufferListp x,1 .x∘0∘0))
```

The rule for computing the support for this predicate is

```
(UNION (Support .x) (Support y))
```

This rule is correct if it can be shown to cover the support of the definition. The support of the definition is

```
(UNION (Support (Subset x Bufferspace))
       (Support y≠0)
       (Support (y is less than 37777Q))
       (Support (LENGTH x)=0)
       (Support y=ZERO/MEM)
       (Support (PacketBufferp x))
       (Support y=x∘0∘0/MEM)
       (Suuport (PacketBufferListp x,1 .x∘0∘0)))
```

The first seven clauses simplify to

```
(UNION (Support x) (Support y))
```

The last clause makes use of the rule being tested and produces

```
(UNION (Support .x,1) (Support .x∘0∘0))
```

Altogether, it thus required to show that

```
(UNION (Support x)
       (Support y)
       (Support .x,1)
       (Support .x∘0∘0))
```

is a subset of

```
(UNION (Support .x) (Support y))
```

The rule for computing the support of a "dotted" form is just to take the union of the structure of form under the dot with the support of the form under the dot. (This latter part is required because the form may have other dots nested deeper.) For the cases at hand, we have

```
(Support .x) = (Structure x) U (Support x)
(Support .x,1) = (Structure x,1) U (Support x)
(Support .x∘9∘0) = (Structure x∘0∘0) U (Support x)
```

BEST AVAILABLE COPY

Since (Structure x,1) and (Structure x∘0∘0) are both subsets of (Structure x), the proof is complete.

2.6 Machine descriptions revisited

As described above, the plan is represent the basic machine in terms of a set of SDs and to use these SDs to prove facts about the operation of the IMP code. The original set of SDs to represent the machine must be invented by the user and input to the system.

While it is feasible for the user to write his own machine descriptions using SDs, Bell and Newell have already pioneered the machine description area and invented a quite reasonable notation, ISP. More recent work by Barbacci, Barnes, Cattell and Siewiorek has evolved the language and provided tools for manipulating descriptions written in ISPS, the current derivative of ISP.[10]

I experimented with ISPS and wrote the following description of the Honeywell 316 in ISPS. Only a skeleton of the input-output structure is given, but the rest of the description is intended to be complete.[11]

[10] The original ISP notation is documented in *Computer Structures: Readings and Examples*, by C. Gordon Bell and Allen Newell, published by McGraw-Hill Book Company, New York, 1971. The most recent description of ISPS is an internal Carnegie-Mellon University report, "The ISPS Computer Description Language," by Mario Barbacci, Gary Barnes, Rick Cattell and Daniel Siewiorek, dated August 14, 1977 and available from the Computer Science Department at Carnegie-Mellon University.

[11] The bulk of this description comes from the *Programmers Reference Manual: DDP-516 General Purpose Computer*, published by Honeywell Inc., Framingham, Massachusetts, 1968. However, a number of details were not clear in the manual and I asked for assistance from the IMP crew at BBN. Most of the questions I asked were answerable immediately from practical experience with the hardware. A few questions, however, required experimentation with the machine to see how it would behave. These questions arose just from the attempt to prepare a formal description of the machine. The fact that this exercise forced these details to be made explicit suggests that formal description of computers may be beneficial to architecture designers and technical manual writers independent of any automatic processes that may be applied to the descriptions. I am indebted to entire IMP crew at BBN for their assistance in preparing this description and their patience and responsiveness is ferreting out the details of how the machine behaves under various unlikely sequences of instructions.

```
H316 := (

☆☆ Mp.State ☆☆

 mem[0:#77777]<15:0>,
          x<15:0> := mem[0]<15:0> !Index register is cell 0

☆☆ Pc.State ☆☆

     c<>,         !carry bit
     a<15:0>,     !accumulator -- referred to below as A
     y<14:0>,     !internal register -- holds effective address
          ea<14:0> := y<14:0>, !another name for same
     m<15:0>,     !internal register -- holds word fetched from mem
     op<3:0>,     !internal register -- hangs onto op field
     exx<>,       !internal register -- hangs onto index bit
     b<15:0>,     !extension of accumulator
     pc<14:0>,    !program counter
     sc<5:0>,     !shift counter -- used only for shifts, OTK and INK
     xa<>,        !extend mode option:
                  !1 => extended addressing hardware exists,
                  !0 => not
     sextf<>,     !0 => disable extended addressing at next JMP
     extmd<>,     !1 => in extended addressing mode, 0 => not
     pmi<>,       !previous mode indicator for extended addressing --
                  ! set by interrupt and read by INK
     pi<>,        !1 => interrupts are permitted, 0 => not
     spi<>,       !1 => enable interrupts after next instruction
     inten<15:0>, !vector of enable/disable bits for devices
     z<>          ! a source of zero bits for the shift instructions

☆☆ External.Pc.State ☆☆

 intrq<15:0>, !vector of interrupt requests, set by devices
 ss1<>,  !sense switch 1
 ss2<>,  !sense switch 2
 ss3<>,  !sense switch 3
 ss4<>   !sense switch 4
```

BEST AVAILABLE COPY

```
** Effective.Address.Calculation **

bumppc() :: (Decode extmd => (pc<13:0>←pc+1, pc←pc+1)),

dxea() := (If m<14> => (ea←(ea+x)<13:0>) next
           If m<15> => (m←mem[ea] next y←m next Loop dxea)),

exea() := (Decode m<15> => (0 := (if m<14> => ea←ea+x),
                            1 := (m←mem[ea] next y←m next Loop exea))),

effaddr() := ( Decode extmd => (dxea(), exea())),

xeffaddr() := ( Decode extmd => (
      0 := (If m<15> => m←mem[ea] next dxea()),
      1 := (If m<15> => m←mem[ea] next Loop xeffaddr)))


** Instruction.Execution **

interrupts() := (
     ** Internal.Registers **

     i<15:0>,
     imsb<14:0>,
     xmsb<15:0>

     ** Internal.Procedure **

     priority() := (If i and xmsb eql 0 => (
        imsb←imsb+1 next
        xmsb←xmsb<15:1> next
        loop priority))

     ** Main.Routine **

     Start.Main() := (
        i←inten and intrq next
        If pi and (i neq 0) => (
           (imsb←#64; xmsb←#100000 next priority());
           (spi←0; pi←0; pmi←extmd; sextf←xa next extmd←xa) next
           ea←mem[imsb] next
           intrq←intrq xor xmsb next
           jst()))),
```

BEST AVAILABLE COPY

```
fetch() := (
   m←mem[pc] next                          !Fetch instruction
   exx←m<14>; op←m<13:10>; y<8:0>←m next !Save index bit and opcode
   !Extend address with either 0 or high-order bits of pc,
   !according to page bit
   Decode m<9> => (y<14:9>←0, y<14:9>←pc<14:9>) next
   bumppc()),

generics0() := (Decode m<9:0> => (
  //11\EXA := (sextf←xa; extmd←xa),        !Enter extended addressing mode
  //13\DXA := sextf←0,                     !Leave extended addressing mode
  //43\INK := (a<15>←c;                    !Input keys
             a<14>←undefined();
             a<13>←pmi;
             a<12:5>←0;
             a<4:0>←sc),
  //201\IAB := (a@b←b@a; sc←undefined()), !Interchange A and B
  //401\ENB := spi←1,                      !Enable interrupts
  //1001\INH := (pi←0; spi←0))),          !Inhibit interrupts

shift.loop() := (
   If sc EQL 0 => (leave shift.loop) next
   Decode m<9:6> => (
     0\LRL   := a@b@c←a@b,                    !Long right logical
     1\LRS   := a<14:0>@b<14:0>@c←a@b<14:0>,  !Long right arithmetic
     2\LRR   := a@b@c←b<0>@a@b,               !Long right circular
     3\S403  := undefined.action(),           !Non-existent
     4\LGR   := a@c←a,                        !Logical right
     5\ARS   := a<14:0>@c←a,                  !Arithmetic right
     6\ARR   := a@c←a<0>@a,                   !Circular right
     7\S407  := undefined.action(),           !Non-existent
     8\LLL   := c@a@b←a@b@z,                  !Long left logical
     9\LLS   := (c←c or (a<15> xor a<14>) next
                 a@b<14:0>←a<14:0>@b<14:0>@z), !Long left arithmetic
    10\LLR   := c@a@b←a@b@a<15>,              !Long left circular
    11\S413  := undefined.action(),           !Non-existent
    12\LGL   := c@a←a@z,                      !Logical left
    13\ALS   := (c←c or (a<15> xor a<14>) next
                 a←a<14:0>@z),                !Arithmetic left
    14\ALR   := c@a←a@a<15>,                  !Circular left
    15\S417  := undefined.action()) next      !Non-existent
   sc←sc+1 next loop shift.loop),
```

BEST AVAILABLE COPY

```
skip() := (If m<9> eqv ((m<8> and a<15>) or
     (m<6> and a<0>) or (m<5> and (a NEQ 0)) or !SLN/SLZ & SNZ/SZE
     (m<4> and ss1) or (m<3> and ss2) or        !SS1/SR1 & SS2/SR2
     (m<2> and ss3) or (m<1> and ss4) or        !SS3/SR3 & SS4/SR4
     (m<0> and c)) => (bumppc())),              !SSC/SRC

generics1400() := (Decoda m<9:0> => (
    #0040\CRA := a←0.                      !Clear A
    #1216\ACA := c@a←a+c,                  !Add carry to A
    #1206\AOA := c@a←a+1,                  !Add one to A
    #140/\TCA := a←-a,                     !Two's complement of A
    #0320\CSA := (c←a<15> next a<15>←0),   !Copy sign
    #0024\CHS := a<15>←Not a<15>,          !Change sign
    #0401\CMA := a←Not a,                  !Complement A
    #0500\SSM := a<15>←1,                  !Set sign minus
    #0100\SSP := a<15>←0,                  !Set sign plus
    #0200\RCB := c←0,                      !Reset carry bit
    #0600\SCB := c←1,                      !Set carry bit
    #1050\CAL := a<15:8>←0,                !Clear left part of A
    #1044\CAR := a<7:0>←0,                 !Clear right part of A
    #1340\ICA := a←a<7:0>@a<15:8>,         !Interchange characters in A
    #1140\ICL := a←a<15:8>,                !Interchange and clear left
    #1240\ICR := a<7:0>@a<15:8>←a<7:0>)),  !Interchange and clear right

generics() := (Decode m<15:14> => (
   0 := generics0(),
   1 := (sc←m: c←0 next shift.loop()), !shifts
   2 := skip(),                        !skips
   3 := generics1400())),

jmp() := ((Decode extmd => (pc<13:0>←ea, pc←ea)) next
           extmd←sextf and xa),    !Jump

lda() := (a←mem[ea]),            !Load A

ana() := (a←mem[ea] and a),      !And to A

sta() := (mem[ea]←a),            !Store A

era() := (a←mem[ea] xor a),      !Exclusive or to A

add() := (c@a←a+mem[ea]),        !Add
```

```
sub() := (c@a←a-mem[ea]),         !Subtract

jst() := (                        !Jump and store
   decode extmd => (0 := (mem[ea]<13:0>←pc next pc<13:0>←ea+1),
                    1 := (mem[ea]<14:0>←pc next pc←ea+1))),

cas() := (                        !Compare A and storage
   m←mem[ea] next
   Decode a tst m => (0\LSS := (bumppc() next bumppc()),
                      1\EQL := bumppc(),
                      2\GTR := pc←pc)),

irs() := (                        !Increment and skip on zero
   m←mem[ea]+1 next
   mem[ea]←m next
   If m EQL 0 => bumppc()),

ima() := (                        !Interchange memory and A
   m←mem[ea] next
   mem[ea]←a next
   a←m),

ocp() := (Decode m<9:0> => (      !Output control pulse
    4 := ocp4(),
    #0101 := c←c,
    #0104 := c←c,
    #0041\TASK := intrq<0>←1)),

sks() := (Decode m<9:0> => (      !Skip if ready line set
    4 := bumppc(),
    #0104 := bumppc(),
    #1777\dummy := undefined.action())),

ina() := (If m<9> => (a←0) next   !Input to A
    Decode m<8:0> => (
    4 := (ina4() next bumppc()),
    #777\dummy := undefined.action())),
ota() := (Decode m<9:0> => (       !Output from A
    4 := (ota4() next bumppc()),
    #0120\SMK := inten←a,
    #1020\OTK := (
     c←a<15>; sextf←a<13>; sc←a<4:0> next
     extmd←extmd or sextf))),

i.o() := (Decode m<15:14> => (
    0\OCP := ocp(),
    1\SKS := sks(),
    2\INA := ina(),
    3\OTA := ota()                !SMK and OTK are special case OTAs)),


ldx.stx() := (Decode exx => (
      mem[ea]←x,                  !Load index register
      x←mem[ea]))                 !Store index register

** Instruction.Interpretation **

Start.Main() := (
   interrupts() next
   pi←spi next
   fetch() next
   Decode op => (0 := generics(),
                 1 := (effaddr() Next jmp()),
                 2 := (effaddr() Next lda()),
                 3 := (effaddr() Next ana()),
                 4 := (effaddr() Next sta()),
                 5 := (effaddr() Next era()),
                 6 := (effaddr() Next add()),
                 7 := (effaddr() Next sub()),
               #10 := (effaddr() Next jst()),
               #11 := (effaddr() Next cas()),
               #12 := (effaddr() Next irs()),
               #13 := (effaddr() Next ima()),
               #14 := i.o(),
               #15 := (xeffaddr() Next ldx.stx()),
               #16 := undefined.action(),       !reserved for MPY
               #17 := undefined.action()) next  !reserved for DIV
   Loop Start.Main))
```

One of the tools provided by the CMU group is a parser which accepts an ISPS description and outputs a parse tree, fully parenthesized in prefix format and available in an ASCII file. Using the facilities of the ARPANET, we have found it very convenient to generate ISPS descriptions at ISI in Los Angeles, ship them over the ARPANET to Carnegie-Mellon University in Pittsburgh, parse the description at CMU, and bring the parse output back to ISI. The whole process takes 10 to 15 minutes.

Charlie Hayden has written a program which accepts the parse tree as input and generates state deltas.[12] The current translation of the full description of the 316 results in 310 state deltas. This is a large number of state deltas, and they were put aside for possible later use. The primary reason for the large number is that a separate state delta is generated for each invocation of a function within an expression. Moreover, distinct values are invented for the fictitious microprogram counter, resulting in a very large number of unreadable, generated symbols.

To a certain extent, this expansion of text as ISPS descriptions are translated into SDs is unavoidable because SDs provide no implicit control structures and all "control" has to be encoded as a set of changes to the state description. Perhaps the most troublesome aspect of this large number of SDs is that it is difficult to design an efficient, automatic strategy for selecting which SD to use for advancing to the next state. With respect to this particular issue, an idea has recently emerged for an alternate representation of the internal state of the machine in terms of what state deltas are applicable instead of giving an explicit label to each internal state. This idea is detailed in chapter six.

[12] During my first experiments with ISPS, I wrote a translator from ISPS parse trees into executable Interlisp code. With the help of the CMU group on ISPS details and some handholding from Marty Yonke on Interlisp details, I was able to put together a rudimentary translator in about three weeks. Charlie's program is much cleaner and outputs both executable code in Bliss and state deltas. The organization of the program permits easy addition of other modules to output code in other languages. Recently, Bill Overman has exercised this possibility by writing a trial version of a PL/I code generator. The cost of each of these efforts has been quite modest and probably would not have been undertaken if the parser were not available.

2.7 Formal basis

One of the classical formal views of computing looks at a computer as a transition function F:S→S, where S is the set of possible state vectors and F is the rule for advancing the computation one step. Components of a state vector are accessed by using the name of the component as an index, e.g., s∘A refers to the A component of the state vector and s∘(MEM∘5) refers to word 5 of the MEM component of the state vector. (In prefix form, these are (SEL s A) and (SEL s (SEL MEM 5)), respectively. By treating SEL as an associative operator, the latter expression is equivalent to (SEL (SEL s MEM) 5), which corresponds to (s∘MEM)∘5).

2.7.1 State deltas

One way to look at state deltas is as a shorthand for a specialized class of formulas involving state vectors and transition functions. The first specialization is the suppression of any explicit representation of state vectors. The "." operator takes a name and treats it as an index into the *current* state vector. Within the proof system, only one state vector is "current", and every occurrence of a "." outside of a SD is understood to represent access into this state vector.

The second specialization is the suppression of any explicit representation of the transition function. In place of statements about F, state deltas represent statements about the *closure* of F. The closure of F is defined by

$$F^*(s) = \{F^i(s) : i \geq 0\},$$

that is,

$$F^*(s) = \{s,\ F(s),\ F^2(s),\ F^3(s),\ F^4(s),\ \ldots\ \}.$$

If P' and Q' are the pre- and postcondition of a SD with the "."'s replaced by indexed accesses of s, the SD states that

$$(\forall s)(s \in S \wedge P'(s) \rightarrow (\exists s')(s' \in F^*(s) \wedge Q'(s'))).$$

In common sense terms, this means that state deltas say that the postcondition will be true sometime in the future, but nothing is said about exactly how many steps are needed to reach such a state. While we normally assume that the SDs that we use as an axiomatic description of the hardware are somehow fundamental or atomic, nothing in our theory or proof system can know whether a particular SD represents an atomic step or a long sequence of steps. This limitation simplifies the theory by providing uniform treatment of all computational statements, but it fails to provide a basis for certain classes of arguments in which it is important to know all of the possible states that the machine might be in.

The third specialization concerns the machinery to relate s′ to s. The modification list, M, shows which components of s′ may have values which are different from the same components of s. By implication, components of the state vector which haven't changed must be the same. The statement above thus requires amendment to show that s and s′ are the same except for the components listed in the modification list. Because component names are permitted to overlap, the precise relationship between s and s′ has to be stated in terms of *disjoint* indices. If we use ⊗ to stand for *disjoint*, the statement that s′ is the same as s except possibly at places listed in M is written

$$\forall i(\forall j(j \in M \to i \otimes j) \to s' \circ i = s \circ i).$$

Thus,

```
(SD (pre: P) (mod: M) (env:) (post: Q) (vars:))
```

is an abbreviation for

$$\forall s(s \in S \wedge P'(s) \to \exists s'(s' \in F^{*}(s) \wedge Q'(s') \wedge \\ \forall i(\forall j(j \in M \to i \otimes j) \to s' \circ i = s \circ i))).$$

2.7.2 The proof system

The proof system detailed in the next chapter provides the machinery for deriving new state deltas as theorems using a given set of state deltas as axioms. From the point of view of a state delta as an abbreviation of a formula involving state vectors and transition functions, the basic design of the proof system is simply the following.

When a proof is begun, the SD to be proven is given. The precondition of the SD, P′(s), is assumed to be true, and the goal is to prove

$$\exists s' (Q'(s') \wedge \forall i (\forall j (j \in M \rightarrow i \otimes j) \rightarrow s' \square i = s \square i)).$$

The proof steps that are used in the course of the proof fall into two basic categories. One category is normal derivation in which formulas are combined using the normal inference rules to derive additional formulas. The second kind of step is one which advances the computation. In terms of state vectors, this means that a SD is used to derive a fact about a future state vector s'. In order to keep the bookkeeping quite simple, a rule is imposed that all of the formulas in the proof must refer to the same, "current", state vector. Thus, when a SD is used to derive a fact about a later state vector, all of the formulas in the proof must be checked for consistency with the new state vector. This checking consists of nothing more than an examination of the support of the formula to see if substitution of s' for s would be valid. If so, the formula is retained. If not, the formula is deleted. Deletion of facts from the proof system may result in difficulty in proving a true theorem, but it cannot result in inconsistency.

The formulas that are valid when s' is substituted for s are not actually manipulated because they do not actually contain occurrences of s. Wherever s would be expected to occur, "." occurs instead. Thus, "." serves as a kind of pronoun for the current state vector, and all formulas that remain valid when the state is updated from s to s' are simply left intact. The interpretation of the "."s in the formulas simply changes.

For straightline code, all that is required is that a sufficient number of SDs be applied to derive the postcondition in the SD being proven. For loops, induction is required. Normally, this induction will be carried out using the natural numbers, but any well-founded set may be used. The usual form of a SD that covers a loop is

$$(\forall i \geq 0)(\text{SD (pre: } P(i)) \text{ (mod: } M) \text{ (post: } Q)).$$

The postcondition of this SD specifies what the situation is when the loop is finished, independent of the number of times the loop was executed. The precondition, however, specifies the situation at the top of the loop in terms of the number of iterations yet to go. The method for deriving this SD is to derive two simpler SDs and then use standard mathematical induction rules to derive the form above. The simpler SDs are

$$(\text{SD (pre: } P(0)) \text{ (mod: } M) \text{ (post: } Q)) \text{ and}$$
$$(\forall i \geq 0)(\text{SD (pre: } P(i+1)) \text{ (mod: } M) \text{ (post: } P(i))).$$

The first SD describes the behavior of the system when no more iterations are left and serves as the initial step in the induction. The second SD describes the behavior of the system as it makes one step through the loop and serves as the increment step in the induction.

2.8 Comparison with other formulations

State deltas are related to a number of earlier attempts to formalize the effects of a computational process, particularly McCarthy's "fluents", Fikes and Nilsson' operators in STRIPS, Hoare's axiom system, Igarashi, London and Luckham's "frame axiom", and Manna and Waldinger's "intermittent assertions".

2.8.1 Fluents

John McCarthy considered the problem of how to represent chains of reasoning involving cause and effect relationships. In his memo entitled "Situations, Actions and Causal Laws",[13] he introduced the idea of a predicate which takes a "situation" as an extra argument. The situation argument plays essentially the same role that the state vector s plays in the preceding section. McCarthy also introduced an abbreviation using a similar device of factoring out the situation argument. Predicates which implicitly depended upon a situation were called "fluents".

In McCarthy's formulation, fluents were connected in sentences either by ordinary sentential connectives or by a special operator, "cause". The cause operator is quite close to the idea of a state delta, but McCarthy never made clear how to keep track of which fluents referred to which situations. An additional difficulty is no machinery was provided for removing facts which ceased to be true.

[13] The memo is reprinted in *Semantic Information Processing*, edited by Marvin Minsky, MIT Press, Cambridge, Massachusetts, 1968, pp. 410-417. A later paper with P. J. Hayes titled "Some Philosophical Problems from the Standpoint of Artificial Intelligence," in *Machine Intelligence 4*, edited by B. Meltzer and D. Michie, American Elsevier Publishing Co., Inc., New York, 1969, pp. 463-502.

2.8.2 STRIPS

In the context of building a robot which can solve problems such as moving objects from one room to another, Richard Fikes and Nils Nilsson designed the STRIPS problem-solving system.[14] In the STRIPS system, actions are represented by operators which are composed of a precondition, an add list and a delete list. They serve essentially the same role as our precondition, postcondition and modification list, but the delete list seems to be structured differently. Elements on the delete list usually specify which predicates to delete, compared with our formulation of specifying which places no longer hold the same values and thus searching for all predicates dependent upon the old information.

That difference aside, the STRIPS operators and the state deltas developed here are quite similar. The biggest difference comes in the application. In the STRIPS system, the primary focus is how to build a system which will invent programs composed of the STRIPS operators. In contrast, the work here focuses only on how to represent a sequence of actions. The invention process is assumed to be a separate problem.

2.8.3 Hoare's axiom system

Turning our attention to formulations specifically oriented toward program verification, we see that state deltas bear some resemblance to Hoare's systems of axioms.[15] In Hoare's system, actions are represented as P{S}Q, where P is the precondition, Q is the postcondition, and S is a segment of program code. Although this notation is similar to ours and each lends itself to reasoning about sequential code by simply matching up the postcondition of one predicate with the precondition of the next, there are several differences.

1. With state deltas, termination is included. There is no possibility that

[14] See "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", in *Artificial Intelligence*, Vol. 2, Nos. 3 and 4, North-Holland Publishing Co., Amsterdam, 1971, pp. 189-208, and a later paper by Fikes, Peter Hart and Nilsson, "Learning and Executing Generalized Robot Plans," in *Artificial Intelligence*, Vol. 3, 1972, pp. 251-288.

[15] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, Vol. 12, No. 2, pp. 576-583.

the system will fail to reach state Q. In Hoare's notation, all that is implied is that *if* the system finishes the stated computation, ***then*** Q will be true.

2. With state deltas, the segment of program executed is implicit in the precondition and is not explicitly listed. No distinction is made between control and data. In Hoare's notation, the segment of the program that is executed is listed explicitly. The flexibility provided by my notation permits the user to move the boundary between "control" and "data" whenever he chooses. Moreover, induction over computation sequences requires no special rules; normal mathematical induction may be applied in all circumstances.

3. With state deltas, the effects of the computation are bounded by the list of places modified.

2.8.4 The frame axiom

In their axiom system for characterizing program behavior, Igarashi, London and Luckham extend Hoare's rules to a variety of syntactic forms found in programming languages.[16] For procedures, a "frame axiom" is introduced. The content of their frame axiom is simply if P is a predicate which is true before procedure Proc is entered, and if P and Proc have no variables in common, then P will continue to be true when Proc is exited. This axiom provides some leverage for reasoning separately about the part of the state that has changed during a segment of computation and the part that has not. However, the axiom is restricted to procedure calls and is not applicable to bodies of loops or alternative paths in a conditional statement. At present, I don't believe that any program verification system based on the Floyd-Hoare system makes use of the frame axiom or anything similar to avoid processing the entire state vector at every juncture.

[16] Shigeru Igarashi, Ralph London and David Luckham, "Automatic Program Verification I: A Logical Basis and Its Implementation," *Acta Informatica*, Vol. 4, No. 2, pp. 145-182.

2.8.5 Intermittent assertions

More recently, Manna and Waldinger have taken an idea presented by Burstall and constructed the notion of *intermittent assertions*.[17] Intermittent assertions are very close in spirit to state deltas. For higher level language programs in which the code is pure and effectively disjoint from the data, intermittent assertions correspond to state deltas in which the environment clause points to just the code and the modification list points to all of the data. Intermittent assertions provide the same power to prove termination as state deltas, but provide no difference in representation of state information from the usual inductive assertion technique introduced by Floyd.

The term "intermittent" refers to a slightly different point of view about execution histories. In their treatment of intermittent assertions, the pre- and postconditions are divided into two parts. One part is a special predicate *At* which specifies (effectively) the value of the program counter. The other component is a general predicate that specifies all of the other relationships that have to hold. Under this dichotomy, Manna and Waldinger take the point of view that the general predicate holds *sometimes* when the program counter has the right value. Accordingly, they say "if at sometime when the program is at L_1 P is true, then sometime the program will be at L_2 and Q will be true." This is fully equivalent with our formulation. For most applications, the general predicate associated with a particular place will always be true when control reaches that point. However, in some applications, different predicates will be true at different times when control reaches the same point in the program. This latter concept is just as easily expressed in the state delta formulation as it is in intermittent assertions, although the point is emphasized more clearly with intermittent assertions.

[17] Burstall's paper is "Program Proving as Hand Simulation with a Little Induction," published in *Information Processing 1974, Proceedings of the IFIP Congress*,North-Holland Publishing Co., Amsterdam, pp. 308-312. Zohar Manna and Richard Waldinger's paper introduces the term "intermittent assertions" and will appear in the *Communications of the ACM*, with the title "Is 'Sometime' Sometimes Better than 'Always'? Intermittent Assertions in Proving Program Correctness".

3. The Proof System

We are now ready to look at the complete structure of the proof system. The purpose of the proof system is to check proofs about the operation of programs or pieces of programs. Statements about the operation of a program are expressed as state deltas, and we can assume that all theorems to be proven are in this form. Since state deltas degenerate into normal conditionals when the *mod* and *env* clauses are empty and all of the predicates in the pre- and postconditions are support-free, the proof system contains all the power of a standard proof system for first-order predicate calculus[18] and could be used for that purpose. However, most of the machinery in this system is geared to the proof of SDs, and we can assume that it is used only to prove SDs.

The general method of proving things is to enter a series of hypotheses into the proof system which define the machine, list the code and define the initial state, and then to enter a series of commands which advance the state of the machine until the final state is reached. Other actions are required for case analysis and induction through loops.

The proof system is divided into two main sections, a *checker* and a *proposer*. The checker maintains a symbolic snapshot of the state of the machine, augmented by various theorems and declarations. The checker is driven entirely by commands it receives from the proposer. The checker examines each command to see if it is well-formed and applicable to the current state. If the command is well-formed and applicable, the checker executes the command by updating its internal state. The only direct output from the checker is a signal back to the proposer indicating whether or not the command worked. However, the checker's lists of known theorems and state information is accessible to the proposer for examination.

The role of the proposer is to suggest reasonable next proof steps. It may do so based on heuristics, by asking the user or by reading a prepared proof from a file. Regardless of where the proof steps originate, the proposer sends each step to the checker to cause the checker to update its internal state.

At the moment, the proposer contains no heuristics. Thus it is simply a small

[18] If the precondition is also empty, the SD is equivalent to the conjunction of the predicates in the postcondition, with whatever quantification is required in the vars clause.

executive routine for obtaining information from the user and/or pulling in prepared proofs from files. Possible extensions to the proposer are discussed in chapter six.

The checker is recursive. During any proof, a new subproof may be started. When that subproof is complete, the newly proven SD is added to the list of known theorems in the original proof.

3.1 Contexts

Within a subproof, the internal state of the checker is called a *context*. The primary components of a context are a list of predicates which describe the current state of the machine, a list of names which are "in use", either as place names, as variables, or both, and a map of the overlap relationships among the various places. Two other lists of places play an important role in connecting lower level contexts to higher level contexts. They are the list of places which may be modified and the environment list.

A new context is created when a new subproof is begun and is destroyed when the subproof is complete. Many of the components of the context are initialized to the current value of the corresponding component of the superior subproof. In principle, the new context's components are copies of the old context's components. However, one of the goals in the design of the system is to minimize the copying of constant information, and with one exception the initial values for the new context are formed simply by setting up a pointer to the current value in the upper context. The exception will emerge in the discussion below.

3.1.1 USABLE

The most important component of a context is the list of accessible predicates. This list is called USABLE. Conceptually, four different kinds of predicates cohabit this list:

1. state predicates

   These predicates constrain the values stored in the places. Since the machine state must satisfy all of these predicates, their conjunction is the current state description.

2. state deltas

These predicates describe what changes to the state may take place by forward execution of the machine. The precondition of the SD determines when the SD is applicable, so the mere appearance of the SD on USABLE does not guarantee that the SD will ever be useful.

3. place relationships

These predicates relate sets of places to each other. The key relationships of interest are that two or more places are disjoint from each other and that a set of places is a decomposition of another place. These two relationships are summarized in the predicate

$$(\mathrm{Covering}\ P_0\ \langle P_1 \ldots P_k \rangle)$$

which states that $P_1 \ldots P_k$ are pairwise disjoint and that each is contained in P_0. Place relationships are also stored in the place graph, described below.

4. general facts

Definitions of basic terms and various lemmas are often needed in the course of a proof. These predicates are not specific to the state of the machine, or even to the notion of computation. For example, the definition of *factorial* or the definition of *ordered* fall into this category.

Since any two predicates on this list may be combined to form a conjunction, these categories are not rigorous. However, the proof system does not try to classify the predicates according to these categories. Each command executed by the checker has its own criterion for applicability.

All predicates on USABLE are cross-referenced according to the places they depend upon. The purpose of the cross-referencing is to be able to find and delete any predicate which depends upon a place which has been modified (or is assumed to have been modified.)

The mechanism for cross-referencing the predicates is the following. Whenever a predicate is added to the USABLE, it is analyzed syntactically to determine the list of places which support it. The intent is that a predicate should be left on the list until the value stored in a supporting place is changed. For example .A=0 is supported by A and .A+.B=0 is supported by A and B. The precise rule for computing support for a predicate was discussed in chapter two.

After computing the support of a predicate which is to be added to USABLE, the predicate is added to USABLE and it is also added to the predicate list of every one of the places in its support. The predicate lists are part of the graph structure maintained for places.

USABLE is implemented as a list of predicate records.[19] Each predicate record has a flag, exp, env and placelist component. The exp component is the actual predicate. placelist is the list of supporting places. flag and env are explained below. The initial value of USABLE is a selected subset of the predicates on the copy of USABLE in the superior context, augmented by the precondition of the SD to be proven. The details of the selection process are discussed below.

3.1.2 FREE

FREE is the set of variable names which appear free in formulas in the proof.[20] When a new subproof is started, any variables in the *vars* clause of the SD to be proven are added to FREE. (They must not appear there beforehand, of course.) Similarly, when a new variable name is assigned to a value through the InstantiateContents command, that name is added to FREE.

3.1.3 Places

One of the goals of this system is to provide an efficient method for treating aliasing and overlap among places. In contrast to most verification systems, we do not assume that different place names refer to different places. However, most place names do, in fact, refer to different places, so we need a way of determining the clash (or potential clash) among place names reasonably efficiently.

[19] "Record" is a data structure in Interlisp. Records have a fixed number of components, accessible by field names. Each component may be accessed and/or modified separately.

[20] In this chapter, "FREE" refers only to a component of the context of a subproof and is completely unrelated to the IMP code. I apologize for the confusion.

The *place system* maintains a database of relations among the various place names and is consulted whenever the checker needs to know all possible overlaps among a set of names. The place system is also consulted whenever it is necessary to check that a set of place names refers to completely disjoint places.

The key data structure in the place system is the *place graph*. The place graph contains two types of nodes, *place nodes* and *family nodes*. Place nodes are connected only to family nodes and family nodes are connected only to place nodes. The arcs are directed and the graph is acyclic, so the notions of "up" and "down" are well defined. Each family node contains exactly one arc going up to a place node, but may contain any number (but at least one) of arcs going down to a place node. Place nodes may have any number of arcs going in either direction, including none.[21]

The place graph encodes relationships of the form

$$(\text{Covering } P_0 \ <P_1 \ \ldots \ P_k>)$$

A relationship of this kind is a *family* and is encoded as a single family node in the place graph. P_0 is called the *mother* of the family and each of the P_i are *daughters*.

Each place node is implemented as a record with the following components:

names — The names of the places associated with this place node. Multiple names are synonyms.

flag — A space for marking whether this node has been seen during a traversal. In between calls to the place system, the flag is NIL.

motherfamilies — A list of family nodes in which this place node is a daughter.

daughterfamilies — A list of family nodes in which this place node is the mother.

[21] The idea of using two types of nodes in the place graph is due to Dave Wile. Prior to his suggestion, I had been struggling with a graph structure with only place nodes and connections among the nodes in the form of lists of lists. After switching to explicit representation of the families, the traversing algorithms became clear. Since the traversing algorithms make use of the family nodes as more than simple lists of lists, explicit representation of the family nodes was a critical step in formulating the ideas.

predlist — The list of predicates (in all contexts) which are supported by this place.

Each family node is implemented as a record with the following components:

flag — A space for marking whether this node has been seen during a traversal. In between calls to the place system, the flag is NIL.

motherplace — The place node for the mother of this family.

daughterplaces — A list of the place nodes for the daughters in the family.

3.1.4 MOD and SUPPRESS

When a subproof is begun, one of the parameters supplied is a list of places which may be modified during the course of the proof. This list is accessible during the proof as MOD. Commands which attempt to modify the contents of a place first check MOD. When the subproof is complete, MOD becomes the *mod* clause in the proven SD.

When a subproof is started, the predicates that are accessible in the new context include a subset of the predicates from the higher context. If these predicates are constant for the life of the subproof, no copying is required and they may be accessed directly. However, predicates which are attached to places that may be modified must be moved out of the way and restored when the subproof is complete; this applies to all predicates attached to modifiable places irrespective of whether these predicates will be used at the lower level.

The predicates are moved out of the way by adding them to a list called SUPPRESS. When the subproof is complete, the predicates on SUPPRESS are put back onto USABLE. In the current implementation, predicates are not physically removed from USABLE during this process; a flag attached to the predicate is set to mark it as unavailable. When the subproof is finished, the flag is reset. This mechanism was adopted to permit the proposer to keep pointers into the list of accessible predicates; when the "suppressed" flag is set, routines in the proposer would know that that predicate is not actually available, although it will be available later when the prior context is restored. The proposer does not yet take advantage of this capability.

3.1.5 ENV

In addition to the list of places which may be modified, another list of places is also supplied when a subproof is begun and maintained during the proof as ENV. ENV is a list of places whose values are brought down from the immediately superior context and made available in the new context. Since values are represented only by the predicates on USABLE which reference them, "bringing down the values" translates into bringing down the predicates whose support is entirely contained within ENV. Predicates whose support is partly but not wholly within ENV are not brought down. Note that predicates which do not depend upon any place are always brought down, even when ENV is null.

If ENV and MOD are disjoint, predicates on USABLE in the superior context whose support is entirely contained within the places on ENV are not actually copied to the new context. Whenever they are referenced, the place system is called to see that the higher level predicates are legally visible in the lower context.

If ENV and MOD intersect, predicates whose support is entirely contained within ENV and whose support is at least partially within MOD are actually copied to the new context and the flag component of the original copy of the predicate is set to T to mark the predicate as inaccessible. When the subproof is complete, the flag is reset to NIL, and the predicate becomes accessible in its own context again.

Further details of the algorithms which bring down predicates according to ENV are given below.

When complete, the ENV becomes the *env* clause of the returned SD.

3.1.6 Ancillary components

Two additional components complete the implementation of a context. SDGOAL holds a copy of the entire SD to be proven and GOAL holds a copy of the postcondition of the SD to be proven. SDGOAL is not referenced during the course of the subproof. When the subproof is complete, the SD that is added to the superior context comes from SDGOAL.

GOAL is used only slightly during a subproof. When the Close command is executed, USABLE is examined to see if the postcondition stored in GOAL is currently true.

At some future point, it is contemplated that the proposer may be able to make use of GOAL to determine which command to send to the checker.

3.2 The checker

Commands from the proposer fall into five main classes: opening or closing subproofs, advancing the computation, adding new place relationships, combining aspects of the current state, adding definitions of new terms.

The checker is divided into three subcomponents, the *kernel*. the *place system* and the *simplifier*. The kernel receives the commands from the proposer, checks each one for applicability and makes the appropriate changes to the context. Whenever one of these actions requires knowledge of the relationships among the places, the place system is consulted. Whenever a new predicate is added to the state description or whenever the kernel needs to know if a predicate is true, the simplifier tries to reduce the predicate.

The remainder of this section describes the commands currently implemented. For each command, there is a series oi conditions which are checked before any action is taken. If the command is not recognized or if one of the checks fails, no action is taken and the kernel returns NIL. If the checks succeed, the context is modified (or created or destroyed) according to the rules given below. The kernel then returns with a non-NIL response.

3.2.1 Opening and closing subproofs

A new subproof is initiated by the (Open *pre mod env post vars*) command, where *pre, mod, env, post,* and *vars* are the components of the new SD to be proven.

The variables in the *vars* clause must not be in use, i.e., none of them may be members of FREE. The places in *mod* must all be registered in the place system. If either of these checks fails, no action is taken and NIL is returned.

After these checks are passed, a new context is created.

The old value of FREE is saved and then FREE is augmented with the variables in *vars*.

The old value of MOD is saved and then MOD is set to *mod*.

The old values of SUPPRESS and USABLE are saved and then SUPPRESS is set to NIL.

ENV is augmented by *env* and this new value is put at the top of ESTACK. (OMEGA) is then pushed onto ESTACK.

All predicates which are supported by any of the places listed in *mod* or by places which overlap with *mod* are added to SUPPRESS and marked as suppressed.

Predicates on SUPPRESS whose support is entirely contained within the environment are added to USABLE.

Predicates in *pre* are also added to USABLE.

A copy of the *pre mod env post* and *vars* clauses is kept in SDGOAL.

Finally, PushPlaceSys is called to prepare the place system for new declarations.

A subproof is terminated by the (Close) command. The only check is whether the predicates in the *pre* clause in the corresponding Open command are all on USABLE. If they are, the old context is restored and the new SD is added to it.

3.2.2 Advancing the computation

Only one rule is implemented for advancing the computation, (ApplyInstSD *patt varlist*). ApplyInstSD instantiates a SD on USABLE and applies it to the current state. *patt* is an Interlisp pattern which is matched against USABLE to find a SD to apply. See chapter five for examples of patterns.

varlist is a list of variables and terms in property list format. The terms are to be substituted for the variables.

Several checks are made before taking any action.

patt must match a SD on USABLE.

varlist must match the vars component of the SD.

All of the substitutions must be proper.

The mod component of the SD must be entirely contained within MOD.

All of the predicates in the precondition must be on USABLE.

If all of these checks succeed, predicates supported by places which overlap with the mod list of the SD are deleted from USABLE and then the predicates in the postcondition are added.

3.2.3 Entering new placenames

Three commands are available for adding new names to the place system. (NewDecomposition (Covering ...)) adds a family to the place graph when the mother place is already registered and the daughter places are not. Each of the daughters is added to the list of all registered places and a family node and a set of place nodes are added to the place graph. The whole relationship is also added to another list to keep track of what additions to the place graph have taken place during this subproof. When the subproof is complete and has been closed, these relationships are deleted and the place system is restored to its previous state.

(NewComposition (Covering ...)) also adds a family to the place graph. In this case, however, the daughters must all be registered and the mother must not be. While the NewDecomposition command requires that the Covering relationship be accessible within the current context, the requirements for the NewComposition command are somewhat stiffer. All of the daughters listed in the covering relationship must be known to be disjoint within the place system. The place system is very conservative about disjointness, and it can happen that a set of places can be proven to be disjoint by manipulation of the (Covering ...) predicates within the proof system, while the algorithms of the place system declare that overlap is possible. For example, if (Covering A <B C>) and (Covering A <D E>) are two relationships that have been entered into the place system using the NewDecomposition rule, the place system will assume that B and C each overlap with D and E. At a later time, it may be discovered that B and D are equal and that C and E are equal, thus eliminating two of the four possible overlaps. The place system, however, cannot accept this information and is committed to believing that all of the overlaps are possible. The reason for this restriction lies in the representation used in the place graph. The place graph consists of a set of connections among the places. The algorithms which search the graph are based on the idea that if another place is reached at any time during a search, then that place may overlap with the original place. Adding new links to the graph could only have the effect of showing that places not previously thought to overlap actually do overlap. Since this particular kind of change implies that predicates which were retained during a computation should have been deleted, even this change is ruled out. In summary, once the place system registers a place, the relationships of that place with all other currently registered places is fixed. NewComposition, therefore, must check that all of the daughters are known *by the place system* to be disjoint.

The last command to enter new place names is (EnterSynonym P_1 P_2). P_1 must already be registered and P_2 must be new. The name P_2 is simply added to the list of names in P_1's node. In an earlier implementation, this facility was not provided. As a consequence, equality relationships were represented in terms of one place covering the other. At certain points in the proof, it became necessary to show that one of the points covered the other, while at other times it was necessary to show the reverse. The structure of the place graph forced an ordering between the places and prevented demonstration that each covered the other.

3.2.4 Normal derivations

The commands in this section derive consequences from the predicates in the current context. The computation is not advanced and the place graph is not affected. Except in the case of the Substitute command, predicates added to the current context are simplified according to the following rules in the next section:

> (CombineCases *$patt_1$ $patt_2$*) takes two SDs and combines them into a single SD. The pre- and postconditions of the new SD are the disjunctions of the pre- and postconditions, respectively, of the two existing SDs. The modification and environment lists are the unions of the modification and environment lists, respectively of the two existing SDs. The *vars* list of the existing SDs must be empty.
>
> (InstantiateContents *placename variable*) adds *.placename=variable* to the current context provided *variable* is new.
>
> (ForSome *varslist pred substlist*) adds (FS *varslist pred*) to the current context provided there is a predicate already in the current context which is equal to the result of substituting the terms in *substlist* for the variables in *varslist*. All of the substitutions must be proper. *substlist* is in property list format.

(Substitute *new old patt*) locates both a predicate which matches *patt* and the predicate *new=old*. The term *new* is then substituted for *old* in the predicate matching *patt* and the result added to the current context without simplification. The old predicate is not deleted.

(SimpleEval *patt*) forces the predicate located by *patt* back through the simplification routines. The most common use for this command is reprocess an if-then-else expression after the "if" part has been specified.

(SwapDOTSEL *exp patt*) substitutes an expression of the form (SEL (DOT *a*) *b*) for an expression of the form (DOT (SEL *a b*)) and then simplifies. *exp* must be an expression of the form (DOT (SEL *a b*)) appearing in a predicate in the current context located by *patt*. The substitution of (SEL (DOT *a*) *b*) usually makes it possible to evaluate the term completely.

(SwapDOTsegthru *exp patt*) is similar to SwapDOTSEL except that *exp* must be an expression of the form (DOT (segthru a b)) and it is replaced by (segthru (DOT a) b).

(Overlimit *exp patt*) examines *exp* to see if it can be shown to be zero. If so, all occurrences of *exp* in the predicate located by *patt* are replaced by zero. *exp* must be of the form (SEL *expl n*), where *n* is an integer and the current context has knowledge that *expl* is less than 2^n.

(Underlimit *exp patt*) expects *exp* to be of the form (segthru *expl n*), where *n* is an integer. If the current context contains knowledge that *expl* is less than 2^n, then *expl* is substituted for *exp* in the predicate located by *patt*.

(Makeindexof *exp patt*) looks for (EQ *c* (SEL *a exp*)) or (EQ (SEL *a exp*) *c*) in the current context. If either form is found, (indexof *c a*) is substituted for *exp* in the predicate located by *patt*.

(Derive *exp*) simplifies *exp* and adds it to the current context. This command directly violates the integrity of the proofchecker, since no checking is performed to make sure that *exp* is derivable from the other predicates in the current context. This command is used as a temporary expedient to skip over derivations that are not yet supported by the proof system but appear (to the user) to be sound. Naturally, proofs containing Derive commands are not considered complete.

3.2.5 Simplification

As noted above, except in the case of the Substitute command, before a predicate is added to the current context, it is simplified. The simplification rules currently in use are the following.

If a form is atomic, no further simplification is attempted.

If a form is not atomic, its arguments are simplified before an attempt is made to simplify the form itself.

If the form is headed by an arithmetic operator and the (simplified) arguments are numbers instead of symbolic expressions, the indicated arithmetic is performed.

If the form is headed by SEL, segthru or segfrom and if the arguments are numbers, the indicated operation is performed. If the first argument is of the form (DOT x), the DOT is moved to the outside and the form is returned; pushing the DOT to the outside minimizes the support.

In some cases involving application of two selection operators, it is apparent that one of them is redundant. The cases that are implemented are

(SEL (segthru x y) z) => (SEL x z), if z≤y;

(segthru (segthru x y) z) => (segthru x w), where w = min(y,z).

If the form is headed by DOT, an attempt is made to evaluate it by looking in the current context. If this fails, but the argument is headed by SEL, segthru or segfrom, this operator and the DOT are commuted and the current context is again consulted. This process is designed to evaluate forms such as (DOT (SEL GFREE 15)) when the current context is holding a value for all of GFREE but not explicitly for GFREE•15. When the process of pushing the DOT in and looking up a value has completed, the resulting form, whether or not an evaluation has been successful, is resimplified in accordance with the above rules for SEL, segthru and segfrom. For forms which were not evaluated, and thus were left with an embedded DOT, resimplification results in the DOT being moved back out. For forms which were evaluated, resimplification selects out the component of interest.

"If" statements are simplified if the antecedent can be simplified to either T or NIL.

Array▫x/Array is simplified to just x. Similarly, Array▫(Place/Array) is simplified to just Place. Note that in the first case, x is a selector into the list Array, while in the second case, Place is a placename.

3.2.6 Definition of new terms

The last set of commands provide a way of introducing new predicates to the proofchecker. Two commands are provided, one for introducing the syntax of the new predicate and one for introducing the definition. If the predicate has only a prefix form and no syntactic extension is required, no command is required for the syntax.

The command for introducing new syntax has the following format.

```
(DefineSyntax newpred (NEWISWORD '(full English singular form)
                                 '(partial English plural form)
                                 '(prefix form)
                                 '(variables)))
```

NEWISWORD is a CLISP function and the body of this command is simply executed. The details on the execution of NEWISWORD are contained in the Interlisp manual. Examples of how this command is used are contained in chapter five.

The command for defining the semantics of a new predicate is

```
(DefineOperator newpred definition support-rule).
```

The support-rule is a pattern used for computing support of forms involving the new predicate. In principle, the support-rule should be checked in accordance with the discussion in chapter two. At present, however, this requirement is not implemented. Examples of the use of this command are also contained in chapter five.

3.3 The place system

The place system is a more or less self-contained set of routines which are called from the kernel at various points in the execution of a command.

3.3.1 FindPlaceNode

FindPlaceNode is the function which translates a place name into a place node. If a simple place name is not registered, NIL is returned. If FindPlaceNode is given a subscripted place name, a pointer to the node corresponding to the smallest registered component is returned. If the base name is not registered, NIL is returned.

Place names are added to the hash table by the kernel commands NewComposition, NewDecomposition and EnterSynonym. A record is kept of each of the actions which causes names to be added to the place list and nodes to be added to the place graph, and all of the actions carried out during the course of a subproof are undone when the subproof is closed. This is a mechanism for permitting names and their overlap relationships to exist just within the scope of a subproof (and its subproofs). PushPlaceSys is the routine called by the kernel when a subproof is begun; PosPlaceSyS is called when the subproof is complete.

3.3.2 Traversing algorithms

(FindAffectedPlaces placelist) returns a list of all of the place nodes which are not guaranteed to be disjoint from the places listed (by name) in placelist. FindAffectedPlaces is called during execution of the ApplyInstSD.

(MarkP p fprime), (MarkFM f) and (MarkFD f) are the basic graph traversing functions.

In a call to MarkP, p is a pointer to place node and fprime is a pointer to the family node, if any, from which p is being reached. MarkP checks flag in the place node. If flag is not NIL, this place node has been visited already and no further action is taken except to set CONFLICTFLAG to T. (CONFLICTFLAG is checked by AllDisjointp; see below.)

If flag is NIL, it is set to T, the place node is added to MARKEDPLACES, MarkFM is called for each of the families listed in motherfamilies and MarkFD is called for each of the families listed in daughterfamilies, *except that the family* fprime *is exempted from exploration*.

MarkFM is called from MarkP with a pointer to a family which contains the place node as a daughter. MarkFM checks flag in the family node. If the flag is already set, no further processing takes place. If flag is not already set, it is now set and (MarkP f:motherplace f) is called to explore the mother of the family and her relatives. The inclusion of f in the calling sequence to MarkP prevents MarkP from attempting to re-explore the graph through this family. f is also added to the MARKEDFAMLIES list for use by UnMark in erasing all of the flags.

MarkFD is called from MarkP with a pointer to a family which contains the place node as a mother. MarkFD checks flag in the family node and terminates if it is already set. If flag is not already set, it is set now and f is added to MARKEDPLACES. MarkP is called for each of the places listed in the daughterplaces component of the family. MarkP is prevented from re-exploring this family by including f in the call to MarkP.

(AllDisjointp placelist) is used by NewComposition to check if all of the places listed (by name) in placelist are known to be disjoint.[22] AllDisjointp is implemented by setting CONFLICTFLAG to NIL, calling MarkP for each of the place nodes corresponding to a place listed in placelist, and returning the value of CONFLICTFLAG. UnMark is called after the graph is traversed to erase all of the flags that had been set.

(EntirelyContainedinp subplacenodes superplacenodes) checks to see if each of the places listed in subplacesnodes is a subplace of at least one of the places listed in superplacenames. EntirelyContainedinp operates by setting flag in each place node corresponding to a place named in superplacenames and then calling Containedinp for each of the nodes listed in subplacenodes. Containedinp explores every upward path either until it encounters a marked place node or until all paths have been exhausted. If a marked place is encountered, Containedinp returns T; otherwise it returns NIL. EntirelyContainedinp returns T if each of its calls to Containedp returned T; otherwise it returns NIL.

EntirelyContainedinp is called to determine if the support for a particular predicate is entirely within the set of places which form the environment for the subproof which is immediately subordinate to the proof in which the predicate was created. If so, the predicate is considered "visible" and is available for use in all subordinate subproofs.

[22] After I first coded Alldisjointp, I re-examined NewComposition and revised it to require only a Covering predicate similar to NewDecomposition. Later, I discovered Alldisjointp was not used anywhere. I let the matter set for a while. Rod Burstall came by and examined the graph structure. As the result of that discussion, I re-examined the algorithms in the place system, and I discovered that I had erred in the implementation of the NewComposition and that Alldisjointp was required as described earlier.

4. A Slice of the IMP Code

While the whole of the IMP code was considered in formulating the theory and building the system, only a small slice of the code was examined in detail. This code is central to the operation of the IMP and posed a large number of theoretical problems. Solving these problems took us much further than I anticipated. While I have not pushed any other sections of the code through the system, I believe that the existing theory and system are sufficient to support proof of much of the rest of the code.

This chapter covers just the slice of code, including a description of the hardware and an overview of the ARPANET. In chapter five, we will return to this code and show its complete proof through the system.

4.1 The ARPANET

The ARPANET is a communication system which connects a number of "host" computers. These host computers send messages to each other through the ARPANET. The ARPANET is implemented as a connected set of IMPs and telephone lines. The IMPs are small computers, predominantly Honeywell 316s, located next to the various hosts. Each host is connected to a single IMP, but one IMP may be connected to more than one host. The IMPs are connected to each other through leased, non-switched telephone lines, usually capable of carrying 50 kilobits per second.

Hosts send messages to other hosts by transmitting each message to its local IMP. Messages are limited to about 8000 bits and begin with a leader which includes the address of the destination host.

The sole role of the IMPs is to move messages between hosts. To accomplish this, messages are broken into smaller units, called packets, and the packets are sent from IMP to IMP until reaching the IMP connected to the destination host. Packets are no longer than about 1000 bits. Messages are subdivided into packets solely for efficiency reasons which are not relevant to the present discussion.

The program within the IMP consists of several interrupt-driven processes. When a packet arrives from another IMP or a message arrives from a host, the modemto-IMP process or the Host-to-IMP process is started to read the packet or message. Messages are broken into packets as they are read in, so we may view the input from the host as just a sequence of numbered packets.

Space for packets is allocated dynamically as needed. There is a fixed number of packet buffers, each capable of holding exactly one packet. Packet buffers are attached to various queues using standard list processing techniques. A packet remains in the packet buffer it was read into until it is transmitted out to another IMP or a host; packets are never copied from one place in core to another.

There is a fixed number of queues in the program: one for each output port, one for packets waiting for routing, the free packetbuffer queue. Every packetbuffer is on just one of these queues or is serving as an input buffer for one of the input ports. Collectively, the queues and the input buffers always account for all of the buffers in the system. The complete set of buffers is determined at assembly time.

4.2 The 316

The Honeywell 316 is a one address, 16-bit minicomputer. In the standard configuration, the memory may have 16K words of 16 bits, thus requiring 14 bits to address each word.

The CPU has a single general purpose register (A) and a program counter (PC). Word zero of memory acts as the index register.

All instructions are one word long, and have one of two formats. Instructions which reference memory have the following format (bit 0 is the least significant bit):

Bit 15	Indirect addressing flag
Bit 14	Indexing flag
Bits 13-10	Operation code (not equal to zero)
Bit 9	Page flag
Bits 8-0	Address field

Because only nine bits are provided in the address field, an instruction cannot address all of memory directly. If the page flag is zero, the address field holds an *absolute address* and thus refers to a cell in the first 512 words of memory.

If the page flag is one, the address field holds a *local address* and thus refers to a cell in the same 512 word page as the instruction, i.e., the high-order 5 bits are the same as the location of the instruction.

If the index flag is set, the contents of cell zero is added to the direct address calculated above.

If the indirect flag is set, the address is used as the location of another address. In the new address word, bit 15 is again the indirect flag, bit 14 is the index flag and bits 13-0 are a full 14 bit direct address. Indirect addressing continues until an address word is encountered with the indirect flag set to zero. Indexing is applied at every step if the index flag is on.

Instructions which do not reference memory are no-address instructions. Bits 13-10 of these instructions are zero and the other bits form an extended operation code.

The specific instructions of concern are the following:

4.2.1 No-address instructions

Clear A register (CRA); 140040Q

The effect of this instruction is to set all sixteen bits of the A register to zero.

Skip Not Zero (SNZ); 101040Q

This instruction tests the A register. If any of the bits are 1, the next instruction is skipped. If the A register holds zero, the next instruction is executed.

4.2.2 Memory referencing instructions

Jump (JMP); opcode = 1

This instruction transfers control to the effective address.

Store A register (STA); opcode = 4

The contents of the A register are copied into the referenced memory location.

Increment and skip (IRS); opcode = 10Q

The value in the referenced memory location is increased by 1 (modulo 2^{16}). If the result is zero, the next instruction is skipped.

Interchange Memory and A register (IMA); opcode = 11Q

This instruction exchanges the contents of the A register with contents of the cell referenced in memory.

4.3 GFREE

We now come to a specific section of code, GFREE. GFREE is called by an input process to allocate a buffer from the free packet buffer queue. Two returns are possible, one indicating success and the other indicating that no free buffers existed.

```
Location   Contents   Label       Source Code      Comments

010511                GFREET:     BSS 1            /LAST BUFFER USED
010512     000000     GFREE:      0                /GET A FREE BUFFER
010513     140040                 CRA
010514     126277                 IMA FREE I       /CLEAR CHAIN PTR
010515     101040                 SNZ
010516     103512                 JMP GFREE I      /NO BUFFERS, NO SKIP
010517     024500                 IRS NFS          /KEEP COUNT
010520     026277                 IMA FREE         /UPDATE FREE LIST
010521     011511                 STA GFREET       /LEAVE A CLUE
010522     025512                 IRS GFREE        /SKIP=SUCCESS
010523     103512                 JMP GFREE I
```

Entry is via a subroutine call instruction which leaves its return address in GFREE and transfers control to the cell after GFREE, 10513Q. If no buffers are available, the return does *not* skip. If successful, the return skips one instruction and the A register contains a pointer to the allocated buffer.

The free list consists of packetbuffers whose first words are strung together in a pointer chain. The last packet buffer header points to a word which contains 0. (This is *not* the same as having the last element point to 0.) FREE points tc the first packet buffer if there is one, or to the dummy packet buffer if the list is empty. No packet buffer begins at word 0.

NFA and NFS contain counters. NFA is incremented whenever a packet is put onto the free list, and NFS is incremented whenever a packet is taken off. The difference between the contents of NFA and NFS gives the actual number of packets on the free list at any time, except in the middle of routines which change them.

As a convenience to one of the callers of GFREE, GFREET also contains a pointer to the allocated buffer if successful.

It is intended that NFS never be incremented up to 0 (thus causing the second IMA to be skipped). The mechanism for preventing this is a background program which periodically stores the difference between the contents of NFA and NFS in NFA, and sets NFS to 0. This routine inhibits interrupts when it does this. Correct operation of this routine is thus timing-dependent. We will avoid this problem by explicitly assuming that the contents of NFS be less than 177777Q when GFREE is entered.

The places which can be modified are PC, A, GFREE, GFREET, NFS, FREE, ZERO and the first word of the first packetbuffer. There are also internal registers which are modified. If we let Q stand for the internal registers and X stand for the whole packetbuffer list which begins at FREE, we can say that nothing outside of the list Q, PC, A, GFREE, GFREET, FREE, ZERO and X is modified.

The final IRS must not skip. It could only skip if the calling location were 177776Q, so that 177777Q would be stored into GFREE upon entry. Addresses in this range would require a 64K address space - far more than the machine has. Consequently, the final IRS cannot skip if GFREE actually holds a legal address.

4.4 Specification of GFREE

Before we can contemplate proving the correctness of GFREE, we need a clear statement of the intended effect of GFREE. One straightforward way to write down the intended effect is to compare the state of the machine before GFREE is entered with the state after it has been executed.

The conditions that are assumed to hold before GFREE is entered are the following:

B1. The program counter holds 105130;

B2. GFREE holds a return address ra, and ra is strictly less than 377770;

B3. The code listed above has not be n modified;

B4. The free packetbuffer list is well-formed, a concept we will expand below;

B5. NFS holds a value nfs which is strictly less than 177770

The conditions that are assumed to hold after GFREE has been executed depending upon whether there were any packetbuffers available on the free packetbuffer list. If there were not, the following conditions are expected:

AZ1. The program counter holds ra;

AZ2. The code listed above has not been modified;

AZ3. The free packetbuffer list is still well-formed but empty;

AZ4. NFS still holds nfs;

AZ5. Only the places listed in item 7 above may have been modified. This restriction is one-sided. It means places not listed have not been modified. It does not gı arantee that listed places were modified.

If there was a packetbuffer available, we expect the following conditions to hold:

AN1. The program counter holds ra+1;

AN2. The code listed above has not been modified;

AN3. The free packetbuffer list is still well-formed and accounts for all but one of the packetbuffers in the original list;

AN4. The A register and GFREET hold a pointer to the other packetbuffer;

AN5. NFS holds nfs+1.

AN6. Only places listed in item 7 may have been modified.

Packetbuffers are packetlength words long and the entire set of possible packetbuffers is determined at assembly time. Bufferspace is the name of that set. None of the packetbuffers begin at word zero in memory. Bufferspace is separate from the code, NFS, GFREE, GFREET, NFS, FREE and ZERO.

A *packetbuffer list* is a finite sequence of packetbuffers with the first word of all but the last packetbuffer holding the address of the first word of the next packetbuffer and the last packetbuffer holding a pointer to a word which contains 0. By convention, this word is the cell ZERO.

A slightly more precise way to characterize packetbuffer lists is the following: X is a packetbuffer list beginning at y if X is a zero-length sequence and y is the address of ZERO or X is a sequence whose length is not zero, the first element of X is a packetbuffer, y is the address of the first word of that packetbuffer and the rest of X is a packetbuffer list beginning at the address contained in the first word in the first packet buffer in X. The list of free packetbuffers is the packetbuffer list beginning at the address contained in FREE.

4.5 Informal proof of the correctness of GFREE

We are now ready to look at the reasons why we believe GFREE performs correctly. Our goal is to show that if the initial conditions B1 through B5 are true at some time, then execution of GFREE leads to either AZ1 through AZ5 or AN1 through AN6. Our technique for achieving this goal is to display the state of the machine, and then step through the program updating the display. This technique is generally called symbolic execution.[23]

We clearly need to display current values for the program counter (PC), A, GFREE, GFREET, NFS, FREE and the free packetbuffer list. The free packetbuffer list presents a problem because we don't know exactly what words in memory are involved. However, we can divide the analysis into two cases, one for an empty list and one for a non-empty list. In both cases the list is represented symbolically by X.

One other detail needs attention. For a careful symbolic execution, we need to assure ourselves that the indirect addressing computed at 105140Q, 105160Q and 105230Q all refer to cells containing numbers strictly less than 1000000Q, i.e., that the indirect

[23] See, for example, "Symbolic Execution and Testing," by James King, IBM Research report, RC 5082.

addressing terminates in just one subcycle. To follow this detail, we'll make use of a fictitious microprogram counter (UPC) and an internal memory address register (M). The UPC holds either top, addr or action. When UPC holds top, the contents of PC is the address of the next instruction. If the instruction is a memory referencing instruction, UPC is set to addr and address fetches take place until a word is accessed with the indirect bit off. At that point, the UPC is set to action and the instruction is executed.

For no-address instructions, execution takes place immediately and the UPC retains the value top.

Here is the display for the initial state. The numbers above the symbols are the memory address. In the case of the free packetbuffer list, p is the address of the first word of the first buffer.

```
Memory cell                          500Q    105120  105110    277Q   p
Name     UPC     PC       M    A     NFS     GFREE   GFREET    FREE   X
Value    top     105130   ?    ?     nfs     ra      ?         p
```

We now begin execution. The instruction at 105130 is CRA, so only PC and A are changed.

```
Memory cell                          500Q    105120  105110    277Q   p
Name     UPC     PC       M    A     NFS     GFREE   GFREET    FREE   X
Value    top     105140   ?    0     nfs     ra      ?         p
```

The PC now points to an IMA instruction. Before proceeding, we need to know whether the free packetbuffer list is empty. We consider the empty case first. Accordingly, the state we are considering is thus:

```
Memory cell                          500Q    105120  105110    277Q   p
Name     UPC     PC       M    A     NFS     GFREE   GFREET    FREE   ZERO
Value    top     105140   ?    0     nfs     ra      ?         y      0
```

The IMA instruction is now partially decoded and discovered to be a memory reference instruction, leading to

```
Memory cell                          500Q    105120  105110    277Q   p
Name     UPC     PC       M    A     NFS     GFREE   GFREET    FREE   ZERO
Value    addr    105140   277Q 0     nfs     ra      ?         p      0
```

277Q is the indirect address. An indirect addressing cycle is taken, leading to

```
Memory cell                          500Q    105120  105110    277Q   p
Name     UPC     PC       M    A     NFS     GFREE   GFREET    FREE   ZERO
Value    action  105140   p    0     nfs     ra      ?         p      0
```

The operation code is now examined and seen to be 11Q indicating an IMA instruction. The contents of cell 277Q is p, and the contents of A is 0.

By convention, p is the address of ZERO, a cell holding 0. After execution of the IMA instruction, the state is:

Memory cell					500Q	10512Q	10511Q	277Q	p
Name	UPC	PC	M	A	NFS	GFREE	GFREET	FREE	ZERO
Value	top	10515Q	?	0	nfs	ra	?	p	0

Interpretation of the instruction at 10515Q now takes place. This is a SNZ instruction, which does not skip. The new state is:

Memory cell					500Q	10512Q	10511Q	277Q	p
Name	UPC	PC	M	A	NFS	GFREE	GFREET	FREE	ZERO
Value	top	10516Q	?	0	nfs	ra	?	p	0

The instruction at 10516Q is an indirect jump. After the fetch of the indirect address, the state is:

Memory cell					500Q	10512Q	10511Q	277Q	p
Name	UPC	PC	M	A	NFS	GFREE	GFREET	FREE	ZERO
Value	addr	10517Q	ra	0	nfs	ra	?	p	0

Since ra is an address, the high-order bit of M is off and no further indirect addressing takes place. After the jump, the state becomes:

Memory cell					500Q	10512Q	10511Q	277Q	p
Name	UPC	PC	M	A	NFS	GFREE	GFREET	FREE	ZERO
Value	top	ra	?	0	nfs	ra	?	p	0

This is a final state.

Alternatively, the free list might not have been empty. We return to the state just prior to execution of the IMA instruction and the state we are considering is:[24]

Memory cell					500Q	10512Q	10511Q	277Q	p
Name	UPC	PC	M	A	NFS	GFREE	GFREET	FREE	X
Value	top	10514Q	?	0	nfs	ra	?	p	p_1

Again we complete the indirect address and the state just prior to the actual exchange is:

Memory cell					500Q	10512Q	10511Q	277Q	p
Name	UPC	PC	M	A	NFS	GFREE	GFREET	FREE	X

[24] In the displays which follow, X refers to just the first word of the first packetbuffer instead of the whole packetbuffer list.

Value	action	10515Q	p	0	nfs	ra	?	p	p_1

Note that we needed to know that the high-order bit in FREE was zero in order to terminate the indirect addressing cycle. After execution of the exchange, the new state is:

Memory cell					500Q	10512Q	10511Q	277Q	p
Name	UPC	PC	M	A	NFS	GFREE	GFREET	FREE	X
Value	top	10515Q	?	p_1	nfs	ra	?	p	0

The A register now holds p_1. p_1 cannot be zero, so the SNZ instruction at 10515Q will skip. After its execution, we have:

Memory cell					500Q	10512Q	10511Q	277Q	p
Name	UPC	PC	M	A	NFS	GFREE	GFREET	FREE	X
Value	top	10517Q	?	p_1	nfs	ra	?	p	0

The IRS instruction at 10517Q is now executed. nfs is small enough to prevent overflow, so we arrive at:

Memory cell					500Q	10512Q	10511Q	277Q	p
Name	UPC	PC	M	A	NFS	GFREE	GFREET	FREE	X
Value	top	10520Q	?	p_1	nfs+1	ra	?	p	0

The instruction at 10520Q is the second IMA instruction. It just causes the contents of A and FREE to be interchanged.

Memory cell					500Q	10512Q	10511Q	277Q	p
Name	UPC	PC	M	A	NFS	GFREE	GFREET	FREE	X
Value	top	10521Q	?	p	nfs+1	ra	?	p_1	0

The next three instructions are quite straightforward. p is put into GFREET, the return address is incremented by 1 and the routine is exited. The fact that ra is a less than 37777Q is needed three times, once to show that the IRS doesn't skip, once to show that the indirect addressing cycle for the jump terminates, and once to show that ra+1 fits into the low-order 14 bits of PC. The final state is:

Memory cell					500Q	10512Q	10511Q	277Q	p
Name	UPC	PC	M	A	NFS	GFREE	GFREET	FREE	X
Value	top	ra+1	?	p	nfs+1	ra+1	p	p_1	0

In the state displays above, each column is dedicated to a specific memory cell or register, and it naturally assumed that each of these cells or registers is completely disjoint from all of the others. As we updated one column in the display, we made the strong assumption that each of the other columns remained valid.

For the registers and fixed memory locations, the independence of each place from all other places is easy to see and doesn't change during execution. For variable places like the free packet buffer list, however, we need some assurance that X is disjoint from all of the other places.

The general situation can be quite complicated. Indirect addressing and list-structured data provide many possibilities for implicit sharing of structures. In order to update the state description correctly, all possible dependencies among the places must be known.

In practice, sharing of places is carefully controlled by the programmer and the number of possible intersections is small. In the IMP code, memory is divided into disjoint regions containing code, fixed data areas and variable data areas. The primary use of the variable data area is for packet buffers. These regions are determined at assembly time and remain fixed for the life of the (particular version of the) IMP code. The variable data area is further divided into packet buffer space and other allocatable storage. The packet buffer space in turn is further structured into individual packet buffers. While we didn't say exactly what cell in memory was named ZERO, we need to know that ZERO is within the fixed data area and hence is *not* within the variable data area. Moreover, ZERO is disjoint from NFS, FREE, GFREE, GFREET, etc. Similarly, we need to know that p is the address of the first word of a packet buffer, and hence cannot be the address of any cell in the fixed data region or the code region. p_1 is also constrained: it may be either an address of the first word of a packet buffer or it may be the address of ZERO, but nothing else.

The assurance that these variable addresses are disjoint from particular other addresses allows us to represent the state of the machine as a set of independent places, each holding its own value, and to update the contents of each place independently of the others. When this assurance is lacking and the contents of a place are modified, the contents of all other places which are not known to be disjoint may have been changed.

5. Proof of GFREE

In this chapter we examine a trace of the proofchecker processing the proof of GFREE. The input to the proofchecker is a series of commands; all of these are included in the trace output. At various points, the trace also includes a list of the predicates added to or deleted from the current context or, in the case of an Open command, the predicates copied into the new context from a superior context. The listing of these predicates is controlled by switches and the points at which these switches are turned on and off is noted in the text. Each command which is input to the system is noted by a preceding bullet (•).

Most of the commands require a reference to an existing predicate within the current context, and I have chosen to use the Interlisp pattern-matching capability to implement this requirement. When a command contains a pattern, the pattern is matched against each of the predicates accessible in the current context until a match is found. Predicates are searched in LIFO order: the most recently created predicate is checked first. If no predicate is found which matches the pattern, the command fails.

Internally, predicates are stored in prefix format. If the predicate .MEM▫(.PC)=1400400Q is entered into the system, it is stored as

```
(EQ (DOT (SEL MEM (DOT PC))) 1400400Q)
```

For the convenience of the user, patterns may be written as if they were to match the external syntax of the predicates. Before the matching process takes place, a pattern is also transformed to an internal prefix format. For example, the pattern &=1400400Q may be used to select the predicate above because this pattern will be transformed into (EQ & 1400400Q) before the match is attempted. ("&" means "match any single element of a list".) For full details on the pattern matching capability, see section 23 of the Interlisp manual.[25] The use of the pattern matcher was a convenient, short-term expediency, and is likely to be changed in future implementations. Some of the problems encountered in using this scheme are discussed in the next chapter.

[25] Warren Teitelman, *Interlisp Reference Manual*, Xerox Palo Alto Research Center, Palo Alto, California, 1975.

5.1 Proof plan

The basic proof of GFREE consists of four sections. The first section steps the machine through the first instruction, CRA. The second section steps the machine through the next two instructions under the assumption that the free buffer list is empty. The third section steps the machine through the second, fourth and following instructions under the assumption that the free list is not empty. The final section combines the results of previous two sections to re-execute the body of GFREE and reach an exit condition. This plan is summarized in the following outline.

```
Begin proof of GFREE
   section 1: execution of first instruction
   Begin proof of GFREE assuming list is empty
     section 2: execution of 2nd and 3rd instructions
     end of proof of empty case
   Begin proof of GFREE assuming list is not empty
     section 3: execution of other instructions
     end of proof of non-empty case
   section 4: single step execution of SD resulting from combination
              of SDs generated by prior subproofs
   end of proof of GFREE
```

Before starting the proof of GFREE, it is necessary to enter into the system a listing of the code for GFREE, a list of notations used in the description of GFREE, a description of the 316 itself, and a few facts about the world that the proof system does not have built in. These add several steps to the outline of the proof. To enter all of the required information, we pretend that we are starting a proof and enter the required information as preconditions. No postconditions will be supplied and the proofs will never be closed. The specification and proof of GFREE therefore take place within several hypotheses and are usable only under the same hypotheses.

```
Begin "proof" introducing general facts about the world
   Begin "proof" introducing description of the 316
      Begin "proof" introducing listing of the code for GFREE
         Begin "proof" introducing notation used in
                       specification of GFREE
            Begin real proof of GFREE
               section 1
               Begin proof of empty case
                  section 2
   end of proof of empty case
Begin proof of nonempty case
   section 3
   end of proof of nonempty case
section 4
end of proof of GFREE
```

The above outline is complete except for one detail. GFREE is specified in terms of a SD that has another SD in its postcondition! In other words, the state delta that describes the operation of GFREE has the following form.

```
(SD (pre: ...)
    (mod: ...)
    (env: ... )
    (post: ... (SD (pre: ...)
                   (mod: ...)
                   (env: ...)
                   (post: ...)
                   (vars: ...))
    (vars: ...))
```

The reasons for specifying GFREE in this form are discussed below. As a consequence, however, an additional level of proof has to be included in our outline. The following is the total outline.

```
Begin "proof" introducing general facts about the world
   Begin "proof" introducing description of the 316
      Begin "proof" introducing listing of the code for GFREE
         Begin "proof" introducing notation used in
                        specification of GFREE
            Begin real proof of GFREE -- outer layer
               Begin proof of inner layer of GFREE
                  section 1
                  Begin proof of empty case
                     section 2
                     end of proof of empty case
                  Begin proof of nonempty case
                     section 3
                     end of proof of nonempty case
                  section 4
                  end of proof of inner layer of GFREE
               end of proof of outer layer of GFREE
```

5.2 General facts

The first command establishes the top level context by postulating some general theorems that are not built into the simplification routines but are necessary at a later point in the proof. In a mature system, there would be a large number of these general facts that would be available and the proof system would be started wi'h these already present. Although these are written as SDs, the pre- and postconditions make no reference to a machine state and the environment and modification lists are empty. Under these conditions, a SD is exactly equivalent to a (universally quantified) conditional statement. Correspondingly, instantiation and application of these SDs is equivalent to instantiating the equivalent quantified conditional statement and then using the *modus ponens* inference rule. Since the machinery to manipulate SDs already existed, we did not bother to build the machinery to handle simple conditional statements.

The five general facts which are supplied state that a list is equal to the concatenation of its first element with the rest of it, that this same concatenation yields a permutation of the original list (because it is in fact identical!), that a number is either zero or not zero, that adding 1 to both sides of an inequality preserves the inequality, and that positive numbers are not zero. @ is the infix operator for concatenation; its prefix form is catenate. The prefix form for is a permutation of is Permutationp. Although the syntax for these operators is predefined, no semantics are built into the system at all.

```
● (Open (pre: (SD (pre:)
                  (mod:)
                  (env:)
                  (post: x=<x▫0>@x,1)
                  (vars: x))
              (SD (pre: x=<x▫0>@x,1)
                  (mod:)
                  (env:)
                  (post: (x is a permutation of <x▫0>@x,1))
                  (vars: x))
              (SD (pre:)
                  (mod:)
                  (env:)
                  (post: (a=0 or a≠0))
                  (vars: a))
              (SD (pre: (x is less than y))
                  (mod:)
                  (env:)
                  (post: (x+1 is less than y+1))
```

```
                (vars: x y))
            (SD (pre: (0 is less than x))
                (mod:)
                (env:)
                (post: x≠0)
                (vars: x)))
    (mod:)
    (env:)
    (post:)
    (vars:))
```

5.3 Definition of the 316

The following Open command establishes a subordinate context in which a small subset of the Honeywell 316 hardware is defined. The SDs that are introduced are identical to the set displayed in chapter two. After the SDs come two declarations about places in the machine. The first declaration introduces the microprogram counter UPC, the program counter PC, memory MEM, internal registers M, I, OP and C, and the accumulator A. These are completely disjoint from each other and partition OMEGA. OMEGA represents all of the space in the machine, so the effect of this declaration is these places are the only places in the machine. The second declaration subdivides memory. In principle, each of the 16,384 cells in memory should be listed separately, but only the cells of interest are explicitly named here. ZERO is a specific cell in memory that is disjoint from all of the other cells listed. In a more complete treatment, it would be introduced as an alternate name for a specifically enumerated cell; this is how the other symbols in the code for GFREE are treated later. Introducing ZERO without disclosing its address illustrates the power of the place graph.

In a similar way, Bufferplace refers to all of the cells in memory that can be used to form buffers. The declaration is again silent about which cells these are, but they are known to be disjoint from ZERO and the other listed cells.

Following the Open command are three commands to build up the place graph. The first two just copy the information provided by the declarations just discussed. The third declaration introduces the name Q as a covering of the internal registers of the machine and adds the covering to the place graph. In the SDs that describe the 316, Q is used in the modification list when it is desired to specify that all of the internal state of the CPU may have changed. Q used in this context is completely separate from the Q used on the end of constants to indicate octal representation.

```
● (Open (pre: (SD (pre: .UPC=top .PC=pc .MEM∘(.PC)=140040Q)
                  (mod: Q PC A)
                  (env:)
                  (post: .UPC=top .PC=(pc+1);13 .A=0)
                  (vars: pc))
              (SD (pre: .UPC=top .PC=pc .MEM∘(.PC)=101040Q)
                  (mod: Q PC)
                  (env:)
                  (post: .UPC=top (if .A=0
                                      then .PC=(pc+1);13
                                     else .PC=(pc+2);13))
                  (vars: pc))
              (SO (pre: .UPC=top .MEM∘(.PC);13,10≠0)
                  (mod: Q)
                  (env:)
                  (post: .OP=.MEM∘(.PC);13,10 .UPC=addr .I=.MEM∘(.PC)∘15
                             (if .MEM∘(.PC)∘9=0
                                 then .M=.MEM∘(.PC);8
                               else .M=.MEM∘(.PC);8+(LOGAND .PC 377000Q)))
                  (vars:))
              (SD (pre: .UPC=addr .I=0)
                  (mcd: UPC)
                  (env:)
                  (post: .UPC=action)
                  (vars:))
              (SD (pre: .UPC=addr .I=1 .M=m)
                  (mod: UPC I M)
                  (env:)
                  (post: .UPC=addr .M=.MEM∘m;13 .I=.MEM∘m∘15)
                  (vars: m))
              (SO (pre: .UPC=action .OP=11 .PC=pc .A=a .M=m .MEM∘m=b)
                  (mod: Q PC A MEM∘m)
                  (env:)
                  (post: .UPC=top .PC=(pc+1);13 .A=b .MEM∘m=a)
                  (vars: pc m a b))
              (SD (pre: .UPC=action .OP=1 .M=m)
                  (mod: Q PC)
                  (env:)
                  (post: .UPC=top .PC=m)
                  (vars: m))
              (SD (pre: .UPC=action .OP=4 .M=m .PC=pc)
                  (mod: Q PC MEM∘m)
```

```
               (env:)
               (post: .UPC=top .PC=(pc+1);13 .MEM∘m=.A)
               (vars: m pc))
            (SD (pre: .UPC=action .OP=10 .M=m .MEM∘m=v .PC=pc)
               (mod: Q PC MEM∘m)
               (env:)
               (post: .UPC=top .MEM∘m=(v+1);15 (if .MEM∘m=0
                                             then .PC=(pc+2);13
                                           else .PC=(pc+1);13))
               (vars: m pc v))
            (Covering OMEGA <UPC PC MEM M I A C OP>)
            (Covering MEM
                    <MEM∘277Q MEM∘500Q MEM∘105110 MEM∘105120
                     MEM∘105130 MEM∘105140 MEM∘105150 MEM∘105160
                     MEM∘105170 MEM∘105200 MEM∘105210 MEM∘105220
                     MEM∘105230 BufferSpace ZERO>))
       (mod: OMEGA)
       (env:)
       (post:)
       (vars: UPC PC MEM M I A C OP Q BufferSpace ZERO))
● (NewDecomposition (Covering OMEGA <UPC PC MEM M I A C OP>))
● (NewDecomposition (Covering MEM
                         <MEM∘277Q MEM∘500Q MEM∘105110 MEM∘105120
                           MEM∘105130 MEM∘105140 MEM∘105150
                           MEM∘105160 MEM∘105170 MEM∘105200
                           MEM∘105210 MEM∘105220 MEM∘105230
                           BufferSpace ZERO>))
● (NewComposition (Covering Q <UPC M I OP>))
```

5.4 Listing of GFREE

The following Open command establishes a further subordinate context in which the instructions which comprise GFREE are entered. In addition, the symbols FREE, GFREE, GFREET and NFS are equated to specific locations in memory. Altogether, this information corresponds to the output from an assembler.

After the Open command comes a set of commands to enter the symbol definitions into the place graph. PureCode is introduced as a name to cover all of the locations containing code which does not change during the operation of GFREE. PureCode will be used in the environment list of the SD which specifies how GFREE operates.

```
• (Open (pre: .MEM∘10513Q=140040Q .MEM∘10514Q=126277Q
              .MEM∘10515Q=101040Q .MEM∘10516Q=103512Q .MEM∘10517Q=24500Q
              .MEM∘10520Q=26277Q .MEM∘10521Q=11511Q .MEM∘10522Q=25512Q
              .MEM∘10523Q=103512Q .ZERO=0 FREE=MEM∘277Q GFREE=MEM∘10512Q
              GFREET=MEM∘10511Q NFS=MEM∘500Q)
        (mod: OMEGA)
        (env:)
        (post:)
        (vars: FREE GFREE GFREET NFS PureCode))
• (EnterSynonym FREE=MEM∘277Q)
• (EnterSynonym NFS=MEM∘500Q)
• (EnterSynonym GFREET=MEM∘10511Q)
• (EnterSynonym GFREE=MEM∘10512Q)
• (NewComposition (Covering PureCode
                            <MEM∘10513Q MEM∘10514Q MEM∘10515Q MEM∘10516Q
                            MEM∘10517Q MEM∘10520Q MEM∘10521Q MEM∘10522Q
                            MEM∘10523Q>))
```

5.5 Additional notation

At this point, definitions for "x is pointing to y", "x is a packet buffer", and "x is bufferlist beginning at y" are introduced. The definitions involve extension to the external syntax as well as postulation of the semantic content of the new terms. These definitions are different from the theorems contained in the top level Open because they are applicable only to the IMP code. The format used here to introduce the syntactic extension is taken directly from CLISP.[26] The semantic definition consists of a name, a predicate which defines the meaning of the name, and rule for computing the support of forms containing the name. If the rule for computing the support is simply the union of the supports of the arguments, the rule need not be specified. In principle, this rule must be justified by an analysis of the definition to show that the list of places computed by the rule at least covers the set of places containing values used in the predicate. This requirement is not implemented, however, and the proofchecker contains a logical flaw until the support rule is forcefully checked.

[26] See chapter 23 of the Interlisp manual.

```
● (DefineSyntax Fointerp (NEWISWORD '(x is pointing to y)
                                    '(are pointing to y)
                                    '(Pointerp x y)
                                    '(x y)))
● (DefineOperator Pointerp ((SD (pre: (x is pointing to y))
                                (mod:)
                                (env:)
                                (post: .x=y/MEM)
                                (vars: x y))
                     and (SD (pre: .x=y/MEM)
                             (mod:)
                             (env:)
                             (post: (x is pointing to y))
                             (vars: x y)))
                  NIL)
● (DefineSyntax PacketBufferp (NEWISWORD '(x is a packet buffer)
                                         '(are packet buffers)
                                         '(PacketBufferp x)
                                         '(x)))
● (DefineOperator PacketBufferp
                  [(SD (pre: (x is a packet buffer))
                       (mod:)
                       (env:)
                       (post: (FS (i) x=MEM,i;(packetlength-1)))
                       (vars: x))
                     and (SD (pre: (FS (i) x=MEM,i;(packetlength-1)))
                             (mod:)
                             (env:)
                             (post: (x is a packet buffer)
                                    (vars: x]
                  NIL)
● (DefineSyntax PacketBufferListp
                (NEWISWORD '(x is a bufferlist beginning at y)
                           'NIL '(PacketBufferListp x y)
                           '(x y)))
● (DefineOperator PacketBufferListp
                  ((SD (pre: (x is a bufferlist beginning at y))
                       (mod:)
                       (env:)
                       (post: (Subsetp x BufferSpace)
                              y≠0
                              (y is less than 37777Q)
```

```
            (if (LENGTH x)=0
                then y=ZERO/MEM
              else x▫0 is a packet buffer and
                                          y=x▫0▫0/MEM
                     and x,1 is a bufferlist beginning
                         at .x▫0▫0))
     (vars: x y))
   and (SD (pre: (Subsetp x BufferSpace)
                 y≠0
                 (y is less than 37777Q)
                 (if (LENGTH x)=0
                     then y=ZERO/MEM
                   else x▫0 is a packet buffer
                          and y=x▫0▫0/MEM
                          and x,1 is a bufferlist
                              beginning at .x▫0▫0))
           (mod:)
           (env:)
           (post: (x is a bufferlist beginning at y))
           (vars: x y)))
<.#1 #2>)
```

5.6 Specification of GFREE

The next Open command introduces the formal specification of GFREE. The specification is written as a compound SD; the postcondition of the top SD contains another SD. The reason for this circumlocution is the need to refer to a variable place which represents the list of free buffers. Whenever a new subproof is begun, the list of places that might be modified needs to be known. All predicates in superior contexts which depend upon any of these places are detached from the database and stored safely away. (Those which are also supported entirely within the environment list are then duplicated and added to the new context.) Since the relationship of the free buffer list to the rest of memory is one of the facts that is listed in the precondition for GFREE, it is necessary to delay using the name of the free buffer list until it can be entered into the place graph. In the precondition of the outer SD, x is introduced as the list of free buffers. When the system tries to compute the support for (PacketBufferListp x .FREE), it finds that x is unregistered and defaults the support to OMEGA.

Prior to beginning the subproof corresponding to the inner SD, x is entered into the place graph. As part of the process of entering a new name into the place graph, the system checks whether any predicates in the current context are supported by OMEGA. If so, the support for these predicates are recomputed. If the recomputed support does not contain OMEGA, the predicate is removed from OMEGA's list and attached to the correct places.

The inner SD contains the meat of the specification. Its modification list contains x along with all the other places that are modified. Since the operation of GFREE depends strongly upon whether or not the the free buffer list was empty when GFREE was entered, the postcondition reflects this difference by appearing as a conditional statement. Notice that final values are specified for ZERO and NFS, even in the cases where these places are not actually modified. These clauses are required because ZERO and NFS are contained in the modification list and are assumed by the proof system to be modified in every execution of GFREE. Were it desired to eliminate the listing of .ZERO=0 from the non-empty case and the listing of .NFS=nfs from the empty case, two separate SDs could be written. Note that .ZERO=0 is required in the empty case because ZERO actually is modified. Restoration of the its value back to its original state is not precisely the same as not changing it all. This point is discussed further in the next chapter.

The environment for the SDs includes PureCode. This provides a terse way to tie the specification of GFREE to the existence of the executable instructions that comprise GFREE in their correct locations. The environment for the inner SD further requires that the values for x, FREE and ZERO not have changed since the inner SD was added to the context. Since ZERO and FREE are also contained on the modification list of the inner SD, it is evident that application of the inner SD will cause its own demise! There is no harm done, however, for the inner SD is added to a context by application of the outer SD and is intended for use exactly once. Reapplication of the outer SD generates a new version of the inner SD, tied to a different instantiation of the free buffer list. Note that the outer SD is not eliminated from a context when the inner SD is applied.

```
• (Open (pre: (x is a bufferlist beginning at .FREE and .ZERO=0))
        (mod:)
        (env: PureCode)
        (post: (SD (pre: .UPC=top .PC=GFREE/MEM+1 .GFREE=ra
                          (ra is less than 37777Q)
                          (0 is less than ra+1)
                          (ZERO/MEM is less than 37777Q)
                          ZERO/MEM≠0
                          (0 is less than nfs+1)
                          .NFS=nfs
```

```
                        (nfs is less than 177777Q))
                  (mod: Q PC A GFREE GFREET FREE x ZERO NFS)
                  (env: PureCode x FREE ZERO)
                  [post: .UPC=top
                         (if (LENGTH x)=0
                             then .PC=ra
                                    and x is a bufferlist beginning at
                                        .FREE
                                    and .ZERO=0 and .NFS=nfs
                           else .PC=ra+1 and .NFS=nfs+1
                                  and (FS (y z)
                                          (x is a permutation of <y>@z
                                             and y is a packet buffer
                                             and z is a bufferlist
                                                 beginning at .FREE
                                             and .ZERO=0
                                             and A is pointing to y∘0
                                             and GFREET is pointing to
                                           y∘0]
                   (vars: ra nfs)))
        (vars: x))

The following preds were added:
pred: (.ZERO=0)
support: ((ZERO))

pred: (x is a bufferlist beginning at .FREE)
support: ((OMEGA))
```

The next two commands take care of the requirement that x be registered in the place system before the proof of the inner SD can be started. The Derive command is a pure, premeditated cheat; it simply adds its argument to the current context without any checking. In principle, the definition of "x is bufferlist beginning at .FREE" provides logical grounds for deriving (Covering Bufferspace <x>), but none of the axioms necessary to carry out this derivation have been added to the system. I do not expect that there will be any difficulty in adding the necessary axioms and in carrying out the necessary derivation.

```
● (Derive (Covering BufferSpace <x>))

The following preds were added:
```

```
pred: (Covering BufferSpace <x>)
support: NIL

● (NewDecomposition (Covering BufferSpace <x>))

The following preds were deleted:
pred: (x is a bufferlist beginning at .FREE)
support: ((OMEGA))

The following preds were added:
pred: (x is a bufferlist beginning at .FREE)
support: ((FREE MEM∘277Q) (x))
```

This Open command begins the subproof of the inner SD. At this point, predicates that are added to the context are no longer shown, but predicates that are copied from a higher context to the new context because of an intersection between the modification list and the environment list are shown.

```
● (Open (pre: .UPC=top .PC=GFREE/MEM+1 .GFREE=ra
              (ra is less than 37777Q)
              (0 is less than ra+1)
              (ZERO/MEM is less than 37777Q)
              ZERO/MEM≠0
              (0 is less than nfs+1)
              .NFS=nfs
              (nfs is less than 177777Q))
        (mod: Q PC A GFREE GFREET FREE x ZERO NFS)
        (env: PureCode x FREE ZERO)
        (post: .UPC=top
               (if (LENGTH x)=0
                   then .PC=ra and x is a bufferlist beginning at .FREE
                            and .ZERO=0 and .NFS=nfs
                 else .PC=ra+1 and .NFS=nfs+1
                        and (FS (y z)
                               (x is a permutation of <y>@z
                                  and y is a packet buffer
                                  and z is a bufferlist beginning at
                                      .FREE
                                  and .ZERO=0 and A is pointing to y∘0
                                  and GFREET is pointing to y∘0)
        (vars: ra nfs))
```

```
The following preds were copied to the new context:
pred: (x is a bufferlist beginning at .FREE)
support: ((FREE MEM∘277Q) (x))

pred: (.ZERO=0)
support: ((ZERO))
```

5.7 Execution of the first instruction

The following commands begin the execution of GFREE. The first command substitutes the address of GFREE into the predicate expressing the current value of the program counter. The simplification rules are bypassed for substitution because they have a tendency to simplify in the wrong direction. In this case, for example, MEM∘105120 is substituted for GFREE in .PC=GFREE/MEM+1. The substitution results in .PC=MEM∘105120/MEM+1. If the simplification rules were invoked, the simplifier would try to "evaluate" .PC and retrieve the value GFREE/MEM+1; the result would be GFREE/MEM+1=105130. While this is a true statement, it does not have the desired effect of changing the representation of the current value of the program counter. Consequently, the unsimplified value is stored away. In the next command, the precondition for the SD includes the condition that .PC=105130. This condition is checked by invoking the simplifier and asking if the expression simplifies to T. At this point the evaluation of .PC works in the right direction. Before the two sides of the equality are compared, the value for .PC is retrieved and automatically reduced to lowest terms. When the equality check takes finally takes place, 105130 is just compared against 105130 and the match succeeds.

All of the switches are turned on at this point. In addition to seeing what predicates are added and removed from the current context, we also see what predicates are found by the pattern matcher.

```
• (Substitute MEM∘105120 GFREE .PC=$)

The pattern matcher returned:
pred: (.PC=GFREE/MEM+1)
support: ((PC))

The following preds were added:
```

```
pred: (.PC=MEM∘105120/MEM+1)
support: ((PC))

● (ApplyInstSD (SD (-- .MEM∘(.PC)=1400400 $) $)
               (pc 105130))

The pattern matcher returned:
pred: (SD (pre: .UPC=top .PC=pc .MEM∘(.PC)=1400400)
          (mod: Q PC A)
          (env:)
          (post: .UPC=top .PC=(pc+1);13 .A=0)
          (vars: pc))
support: NIL

The following preds were deleted:
pred: (.UPC=top)
support: ((UPC))

pred: (.PC=GFREE/MEM+1)
support: ((PC))

pred: (.PC=MEM∘105120/MEM+1)
support: ((PC))

The following preds were added:
pred: (.A=0)
support: ((A))

pred: (.PC=105140)
support: ((PC))

pred: (.UPC=top)
support: ((UPC))

● (ApplyInstSD (SD (-- (x is a bufferlist beginning at y) $) $)
               (x x y .FREE))

The pattern matcher returned:
pred: (SD (pre: (x is a bufferlist beginning at y))
          (mod:)
          (env:)
          (post: (Subsetp x BufferSpace)
                 y≠0
```

```
                (y is less than 37777Q)
                (if (LENGTH x)=0
                    then y=ZERO/MEM
                  else x∘0 is a packet buffer and y=x∘0∘0/MEM
                         and x,1 is a bufferlist beginning at .x∘0∘0))
          (vars: x y))
support: NIL

The following preds were added:
pred: (if (LENGTH x)=0
          then .FREE=ZERO/MEM
        else x∘0 is a packet buffer and .FREE=x∘0∘0/MEM
              and x,1 is a bufferlist beginning at .x∘0∘0)
support: ((x) (FREE MEM∘277Q))

pred: (.FREE is less than 37777Q)
support: ((FREE MEM∘277Q))

pred: (.FREE≠0)
support: ((FREE MEM∘277Q))

pred: (Subsetp x BufferSpace)
support: NIL
```

5.8 Consideration of an empty free bufferlist

The next Open command begins the consideration of the case in which the free buffer list is empty when GFREE is entered. From this point on, only the predicates copied to lower contexts during an Open command and predicates found by the pattern matcher are shown.

```
● (Open (pre: (LENGTH x)=0)
        (mod: x FREE GFREET GFREE A PC Q ZERO)
        (env: OMEGA)
        (post: .UPC=top (LENGTH x)=0 .PC=ra
               (x is a bufferlist beginning at .FREE)
               .ZERO=0 .NFS=nfs)
        (vars:))
```

```
The following preds were copied to the new context:
pred: (if (LENGTH x)=0
           then .FREE=ZERO/MEM
         else x∘0 is a packet buffer and .FREE=x∘0∘0/MEM
                and x,1 is a bufferlist beginning at .x∘0∘0)
support: ((x) (FREE MEM∘277Q))

pred: (x is a bufferlist beginning at .FREE)
support: ((FREE MEM∘277Q) (x))

pred: (.ZERO=0)
support: ((ZERO))

pred: (.UPC=top)
support: ((UPC))

pred: (.PC=10514Q)
support: ((PC))

pred: (.A=0)
support: ((A))

pred: (.GFREE=ra)
support: ((GFREE MEM∘10512Q))

pred: (.FREE is less than 37777Q)
support: ((FREE MEM∘277Q))

pred: (.FREE≠0)
support: ((FREE MEM∘277Q))

● (SimpleEval (if (LENGTH x)=0
                   then $
                 else $))

The pattern matcher returned:
pred: (if (LENGTH x)=0
           then .FREE=ZERO/MEM
         else x∘0 is a packet buffer and .FREE=x∘0∘0/MEM
                and x,1 is a bufferlist beginning at .x∘0∘0)
support: ((x) (FREE MEM∘277Q))
```

The prior command made use of the hypothesis in the Open command. The following ten commands execute the IMA instruction. Four of these ten apply SDs to advance the state. The other six simplify the state to derive the exact precondition required by the SDs. In the future, I expect the proposer to generate these steps automatically.

```
• (ApplyInstSD (SD ($ .MEM∘(.PC);13,10≠0 $) $))
```

```
The pattern matcher returned:
pred: (SD (pre: .UPC=top .MEM∘(.PC);13,10≠0)
          (mod: Q)
          (env:)
          (post: .OP=.MEM∘(.PC);13,10 .UPC=addr .I=.MEM∘(.PC)∘15
                 (if .MEM∘(.PC)∘9=0
                     then .M=.MEM∘(.PC);8
                   else .M=.MEM∘(.PC);8+(LOGAND .PC 377000Q)))
          (vars:))
support: NIL
```

```
• (ApplyInstSD (SD (-- .I=1 $) $)
               (m 277Q))
```

```
The pattern matcher returned:
pred: (SD (pre: .UPC=addr .I=1 .M=m)
          (mod: UPC I M)
          (env:)
          (post: .UPC=addr .M=.MEM∘m;13 .I=.MEM∘m∘15)
          (vars: m))
support: NIL
```

```
• (Substitute FREE MEM∘277Q .I=$)
```

```
The pattern matcher returned:
pred: (.I=.MEM∘277Q∘15)
support: ((FREE MEM∘277Q) (I))
```

```
• (SwapDOTSEL .FREE∘15 .I=$)
```

```
The pattern matcher returned:
pred: (.I=.FREE∘15)
support: ((FREE MEM∘277Q) (I))
```

```
● (Overlimit (.FREE)◦15 .I=$)

The pattern matcher returned:
pred: (.I=(.FREE)◦i5)
support: ((FREE MEM◦277Q) (I))

● (ApplyInstSU (SD (-- .I=0 $) $))

The pattern matcher returned:
pred: (SD (pre: .UPC=addr .I=0)
          (mod: UPC)
          (env:)
          (post: .UPC=action)
          (vars:))
support: NIL

● (Substitute FREE MEM◦277Q .M=$)

The pattern matcher returned:
pred: (.M=.MEM◦277Q;13)
support: ((FREE MEM◦277Q) (M))

● (SwapDOTsegthru .FREE;13 .M=$)

The pattern matcher returned:
pred: (.M=.FREE;13)
support: ((FREE MEM◦277Q) (M))

● (Underlimit (.FREE);13 .M=$)

The pattern matcher returned:
pred: (.M=(.FREE);13)
support: ((FREE MEM◦277Q) (M))

● (ApplyInstSD (SD (-- .OP=11 $) $)
                    (pc 10514Q m ZERO/MEM a 0 b 0))

The pattern matcher returned:
pred: (SD (pre: .UPC=action .OP=11 .PC=pc .A=a .M=m .MEM◦m=b)
          (mod: Q PC A MEM◦m)
          (env:)
          (post: .UPC=top .PC=(pc+1);13 .A=b .MEM◦m=a)
          (vars: pc m a b))
```

```
support: NIL
```

The program counter is now pointing to the SNZ instruction. The next command causes it to be executed. Because the contents of A are known to be 0, the postcondition is completely simplified to just .PC=10516Q. Following the SNZ instruction is the JMP instruction which exits from GFREE. Eight commands are required, four of which advance the computation and four which derive consequences between computation steps.

```
• (ApplyInstSD (SD (-- .MEM∘(.PC)=101040Q $) $)
               (pc 10515Q))

The pattern matcher returned:
pred: (SD (pre: .UPC=top .PC=pc .MEM∘(.PC)=101040Q)
          (mod: Q PC)
          (env:)
          (post: .UPC=top (if .A=0
                              then .PC=(pc+1);13
                            else .PC=(pc+2);13))
          (vars: pc))
support: NIL

• (ApplyInstSD (SD (-- .MEM∘(.PC);13,10≠0) $))

The pattern matcher returned:
pred: (SD (pre: .UPC=top .MEM∘(.PC);13,10≠0)
          (mod: Q)
          (env:)
          (post: .OP=.MEM∘(.PC);13,10 .UPC=addr .I=.MEM∘(.PC)∘15
                  (if .MEM∘(.PC)∘9=0
                      then .M=.MEM∘(.PC);8
                    else .M=.MEM∘(.PC);8+(LOGAND .PC 377000Q)))
          (vars:))
support: NIL

• (ApplyInstSD (SD (-- .I=1 .M=m) $)
               (m 10512Q))

The pattern matcher returned:
pred: (SD (pre: .UPC=addr .I=1 .M=m)
          (mod: UPC I M)
          (env:)
          (post: .UPC=addr .M=.MEM∘m;13 .I=.MEM∘m∘15)
          (vars: m))
support: NIL

● (Substitute GFREE MEM∘10512Q .I=$)

The pattern matcher returned:
pred: (.I=.MEM∘10512Q∘15)
support: ((GFREE MEM∘10512Q) (I))

● (SwapDOTSEL .GFREE∘15 .I=$)

The pattern matcher returned:
pred: (.I=.GFREE∘15)
support: ((GFREE MEM∘10512Q) (I))

● (Substitute ra .GFREE .I=$)

The pattern matcher returned:
pred: (.I=(.GFREE)∘15)
support: ((GFREE MEM∘10512Q) (I))

● (Overlimit ra∘15 .I=$)

The pattern matcher returned:
pred: (.I=ra∘15)
support: ((I))

● (ApplyInstSD (SD (-- .I=0 $) $))

The pattern matcher returned:
pred: (SD (pre: .UPC=addr .I=0)
          (mod: UPC)
          (env:)
          (post: .UPC=action)
          (vars:))
support: NIL

● (ApplyInstSD (SD (-- .OP=1 $) $)
               (m .MEM∘10512Q;13))
```

```
The pattern matcher returned:
pred: (SD (pre: .UPC=action .OP=1 .M=m)
          (mod: Q PC)
          (env:)
          (post: .UPC=top .PC=m)
          (vars: m))
support: NIL
```

At this point, GFREE has been exited. All that remains is to show that the present state matches the postcondition stated in the Open command. The first command reconstructs the fact that x is a bufferlist beginning at ZERO/MEM. The next four transform .PC=.MEM∘10512Q;13 into .PC=ra+1. Finally, .FREE is substituted for ZERO/MEM and the proof is closed.

```
● (ApplyInstSD (SD $ (-- (x is a bufferlist beginning at y)) &)
               (x x y ZERO/MEM))
```

```
The pattern matcher returned:
pred: (SD (pre: (Subsetp x BufferSpace)
                y≠0
                (y is less than 37777Q)
                (if (LENGTH x)=0
                    then y=ZERO/MEM
                  else x∘0 is a packet buffer and y=x∘0∘0/MEM
                       and x,1 is a bufferlist beginning at .x∘0∘0))
          (mod:)
          (env:)
          (post: (x is a bufferlist beginning at y))
          (vars: x y))
support: NIL
```

```
● (Substitute GFREE MEM∘10512Q .PC=$)
```

```
The pattern matcher returned:
pred: (.PC=.MEM∘10512Q;13)
support: ((GFREE MEM∘10512Q) (PC))
```

```
● (SwapDOTsegthru .GFREE;13 .PC=$)
```

```
The pattern matcher returned:
```

```
pred: (.PC=.GFREE;13)
support: ((GFREE MEM∘10512Q) (PC))

• (Substitute ra .GFREE .PC=$)

The pattern matcher returned:
pred: (.PC=(.GFREE);13)
support: ((GFREE MEM∘10512Q) (PC))

• (Underlimit ra;13 .PC=$)

The pattern matcher returned:
pred: (.PC=ra;13)
support: ((PC))

• (Substitute .FREE ZERO/MEM (x is a bufferlist beginning at $))

The pattern matcher returned:
pred: (x is a bufferlist beginning at ZERO/MEM)
support: ((x))

• (Close)
```

5.9 Consideration of a nonempty free bufferlist

The next Open command begins a subproof in parallel with the previous subproof. The modification list for this subproof is the same as before, with the exception of the addition of NFS. All of the clauses copied in the prior subproof are also copied here, along with .NFS=nfs and the SD proven in the prior subproof.

After the proof is opened, the assumption that the list is nonempty is exploited. This will add "x∘0 is a packet buffer", ".FREE=.x∘0∘0/MEM", and "x,1 is a bufferlist beginning at .x∘0∘0" to the context. The support for the last clause will be computed as (x,1 x∘0∘0). Because no subdivisions of x are listed in the place graph, this last predicate will be attached to the node for x and will therefore be subject to deletion if any part of x is modified. Since the next instruction modifies the first word of the first buffer on x, we are in danger of losing all knowledge about the rest of x unless we can subdivide x and break our knowledge about the unmodified part of x into terms that are independent of the modified part.

The next three commands provide the required scaffolding. The result of the Derive command is the subdivision of x we need. The NewDecomposition command enters x∘0 and x,1 into the place graph and the InstantiateContents command gives a name to the current value of .x∘0∘0. All that remains is to separate the knowledge about x into components pertaining to x∘0 and x,1. This is carried out just prior to the actual exchange.

```
● (Open (pre: (LENGTH x)≠0)
        (mod: x FREE GFREET GFREE A PC Q ZERO NFS)
        (env: OMEGA)
        (post: .UPC=top (LENGTH x)≠0 .PC=ra+1 .NFS=nfs+1
               (FS (y z)
                   (x is a permutation of <y>@z
                        and y is a packet buffer
                        and z is a bufferlist beginning at .FREE
                        and .ZERO=0 and A is pointing to y∘0
                        and GFREET is pointing to y∘0)))
        (vars:))

The following preds were copied to the new context:
pred: (.NFS=nfs)
support: ((NFS MEM∘500Q))

pred: (if (LENGTH x)=0
          then .FREE=ZERO/MEM
        else x∘0 is a packet buffer and .FREE=x∘0∘0/MEM
               and x,1 is a bufferlist beginning at .x∘0∘0)
support: ((x) (FREE MEM∘277Q))

pred: (x is a bufferlist beginning at .FREE)
support: ((FREE MEM∘277Q) (x))

pred: (.ZERO=0)
support: ((ZERO))

pred: (.UPC=top)
support: ((UPC))

pred: (.PC=105140)
support: ((PC))

pred: (.A=0)
```

```
support: ((A))

pred: (.GFREE=ra)
support: ((GFREE MEM∘10512Q))

pred: (.FREE is less than 37777Q)
support: ((FREE MEM∘277Q))

pred: (.FREE≠0)
support: ((FREE MEM∘277Q))

pred: (SD (pre: (LENGTH x)=0)
          (mod: x FREE GFREET GFREE A PC Q ZERO)
          (env: OMEGA)
          (post: .UPC=top (LENGTH x)=0 .PC=ra
                 (x is a bufferlist beginning at .FREE)
                 .ZERO=0 .NFS=nfs)
          (vars:))
support: ((OMEGA))

● (SimpleEval (if (LENGTH x)=0
                   then $
                 else $))

The pattern matcher returned:
pred: (if (LENGTH x)=0
          then .FREE=ZERO/MEM
        else x∘0 is a packet buffer and .FREE=x∘0∘0/MEM
               and x,1 is a bufferlist beginning at .x∘0∘0)
support: ((x) (FREE MEM∘277Q))

● (ApplyIns:SD (SD $ (-- x=<x∘0>@x,1) &)
                 (x x))

The pattern matcher returned:
pred: (SD (pre:)
          (mod:)
          (env:)
          (post: x=<x∘0>@x,1)
          (vars: x))
support: NIL

● (Derive (Covering x <x∘0 x,1>))
● (NewDecomposition (Covering x <x∘0 x,1>))
● (InstantiateContents x∘0∘0 p)
```

The IMA instruction is now ready for execution. As before, several commands are required to wade through the details of the indirect addressing cycle. Just before the actual exchange is carried out, the definition of "is a bufferlist" is applied to reduce "x,1 is a bufferlist beginning at .x∘0∘0" to more primitive terms. In the course of the application, all occurrences of .x∘0∘0 are replaced by p because p is known to be the current value of .x∘0∘0. The result is that none of the new clauses are supported by either x or x∘0; only x,1 is needed. Now the exchange instruction may be executed without causing information to be lost.

```
● (ApplyInstSD (SD (-- .MEM∘(.PC);13,10≠0 $) $))

The pattern matcher returned:
pred: (SD (pre: .UPC=top .MEM∘(.PC);13,10≠0)
          (mod: Q)
          (env:)
          (post: .OP=.MEM∘(.PC);13,10 .UPC=addr .I=.MEM∘(.PC)∘15
                 (if .MEM∘(.PC)∘9=0
                     then .M=.MEM∘(.PC);8
                   else .M=.MEM∘(.PC);8+(LOGAND .PC 377000Q)))
          (vars:))
support: NIL

● (ApplyInstSD (SD (-- .I=1 $) (-- m $))
               (m 277Q))

The pattern matcher returned:
pred: (SD (pre: .UPC=addr .I=1 .M=m)
          (mod: UPC I M)
          (env:)
          (post: .UPC=addr .M=.MEM∘m;13 .I=.MEM∘m∘15)
          (vars: m))
support: NIL

● (Substitute FREE MEM∘277Q .I=$)

The pattern matcher returned:
pred: (.I=.MEM∘277Q∘15)
support: ((FREE MEM∘277Q) (I))
```

```
● (SwapDOTSEL .FREE▫15 .I=$)

The pattern matcher returned:
pred: (.I=.FREE▫15)
support: ((FREE MEM▫277Q) (I))

● (Overlimit (.FREE)▫15 .I=$)

The pattern matcher returned:
pred: (.I=(.FREE)▫15)
support: ((FREE MEM▫277Q) (I))

● (ApplyInstSD (SD (-- .I=0 $) $))

The pattern matcher returned:
pred: (SD (pre: .UPC=addr .I=0)
          (mod: UPC)
          (env:)
          (post: .UPC=action)
          (vars:))
support: NIL

● (Substitute FREE MEM▫277Q .M=$)

The pattern matcher returned:
pred: (.M=.MEM▫277Q;13)
support: ((FREE MEM▫277Q) (M))

● (SwapDOTsegthru .FREE;13 .M=$)

The pattern matcher returned:
pred: (.M=.FREE;13)
support: ((FREE MEM▫277Q) (M))

● (Underlimit (.FREE);13 .M=$)

The pattern matcher returned:
pred: (.M=(.FREE);13)
support: ((FREE MEM▫277Q) (M))

● (ApplyInstSD (SD (-- (x is a bufferlist beginning at y) $) $)
             (x x,1 y .x▫0▫0))
```

```
The pattern matcher returned:
pred: (SD (pre: (x is a bufferlist beginning at y))
          (mod:)
          (env:)
          (post: (Subsetp x BufferSpace)
                 y≠0
                 (y is less than 37777Q)
                 (if (LENGTH x)=0
                     then y=ZERO/MEM
                   else x∘0 is a packet buffer and y=x∘0∘0/MEM
                          and x,1 is a bufferlist beginning at .x∘0∘0))
          (vars: x y))
support: NIL
```

```
● (ApplyInstSD (SD (-- .OP=11 $) $ (-- pc m a b))
               (pc 105140Q m x∘0∘0/MEM a 0 b p))
```

```
The pattern matcher returned:
pred: (SD (pre: .UPC=action .OP=11 .PC=pc .A=a .M=m .MEM∘m=b)
          (mod: Q PC A MEM∘m)
          (env:)
          (post: .UPC=top .PC=(pc+1);13 .A=b .MEM∘m=a)
          (vars: pc m a b))
support: NIL
```

The program counter is now pointing to the SNZ instruction. This time, A contains p, and p is known to be different from 0. As a consequence, the postcondition of the SD simplifies completely and .PC=105170.

The four following commands execute the IRS instruction. The only derivation required is to transform .M=500Q into .M=NFS/MEM.

```
● (ApplyInstSD (SD (-- &=101040Q) $ (-- pc))
               (pc 105150))
```

```
The pattern matcher returned:
pred: (SD (pre: .UPC=top .PC=pc .MEM∘(.PC)=101040Q)
          (mod: Q PC)
          (env:)
          (post: .UPC=top (if .A=0
                               then .PC=(pc+1);13
                             else .PC=(pc+2);13))
          (vars: pc))
support: NIL

● (ApplyInstSD (SD (-- .MEM∘(.PC);13,10≠0 $) $))

The pattern matcher returned:
pred: (SD (pre: .UPC=top .MEM∘(.PC);13,10≠0)
          (mod: Q)
          (env:)
          (post: .OP=.MEM∘(.PC);13,10 .UPC=addr .I=.MEM∘(.PC)∘15
                 (if .MEM∘(.PC)∘9=0
                     then .M=.MEM∘(.PC);8
                   else .M=.MEM∘(.PC);8+(LOGAND .PC 377000Q)))
          (vars:))
support: NIL

● (ApplyInstSD (SD (-- .I=0 $) $))

The pattern matcher returned:
pred: (SD (pre: .UPC=addr .I=0)
          (mod: UPC)
          (env:)
          (post: .UPC=action)
          (vars:))
support: NIL

● (Makeindexof 500Q .M=$)

The pattern matcher returned:
pred: (.M=500Q)
support: ((M))

● (ApplyInstSD (SD (-- .OP=10 $) $ (-- m pc v))
               (m NFS/MEM pc 10517Q v nfs))

The pattern matcher returned:
pred: (SD (pre: .UPC=action .OP=10 .M=m .MEM•m=v .PC=pc)
          (mod: Q PC MEM•m)
          (env:)
          (post: .UPC=top .MEM•m=(v+1);15 (if .MEM•m=0
                                                then (.PC=pc+2);13
                                              else .PC=(pc+1);13))
          (vars: m pc v))
support: NIL
```

Because the IRS instruction can skip if the result is equal to zero, it is necessary to show that nfs+1 cannot be zero. The next six commands derive the fact that .NFSffi0 and .PC=105200.

```
• (ApplyInstSD (SD (-- (x is less than y)) $)
               (x nfs y 177777Q))

The pattern matcher returned:
pred: (SD (pre: (x is less than y))
          (mod:)
          (env:)
          (post: (x+1 is less than y+1))
          (vars: x y))
support: NIL

• (Underlimit (nfs+1);15 .NFS=$)

The pattern matcher returned:
pred: (.NFS=(nfs+1);15)
support: ((NFS MEM∘500Q))

• (ApplyInstSD (SD (-- (0 is less than x)) $)
               (x nfs+1))

The pattern matcher returned:
pred: (SD (pre: (0 is less than x))
          (mod:)
          (env:)
          (post: x≠0)
          (vars: x))
support: NIL

• (Substitute .NFS nfs+1 nfs+1≠0)

The pattern matcher returned:
```

```
pred: (nfs+1≠0)
support: NIL

• (Substitute (nfs+1);15 .NFS .NFS≠0)

The pattern matcher returned:
pred: (.NFS≠0)
support: ((NFS MEM∘500Q))

• (SimpleEval (if (nfs+1);15=0
                   then $
                 else $))

The pattern matcher returned:
pred: (if (nfs+1);15=0
          then (.PC=10521Q);13
        else .PC=10520Q)
support: ((PC))
```

The second exchange instruction is now ready for execution. By this time, very little work is required. The key facts are that .A=p and .FREE=x∘0∘0/MEM.

```
• (ApplyInstSD (SD (-- .MEM∘(.PC);13,10≠0 $) $))

The pattern matcher returned:
pred: (SD (pre: .UPC=top .MEM∘(.PC);13,10≠0)
          (mod: Q)
          (env:)
          (post: .OP=.MEM∘(.PC);13,10 .UPC=addr .I=.MEM∘(.PC)∘15
                 (if .MEM∘(.PC)∘9=0
                     then .M=.MEM∘(.PC);8
                   else .M=.MEM∘(.PC);8+(LOGAND .PC 377000Q)))
          (vars:))
support: NIL

• (ApplyInstSD (SD (-- .I=0 $) $))

The pattern matcher returned:
pred: (SD (pre: .UPC=addr .I=0)
          (mod: UPC)
          (env:)
          (post: .UPC=action)
          (vars:))
support: NIL

● (Makeindexof 277Q .M=$)

The pattern matcher returned:
pred: (.M=277Q)
support: ((M))

● (ApplyInstSD (SD (-- .OP=11 $) $ (-- pc m a b))
               (pc 105200Q m FREE/MEM a p b x•0•0/MEM))

The pattern matcher returned:
pred: (SD (pre: .UPC=action .OP=11 .PC=pc .A=a .M=m .MEM•m=b)
          (mod: Q PC A MEM•m)
          (env:)
          (post: .UPC=top .PC=(pc+1);13 .A=b .MEM•m=a)
          (vars: pc m a b))
support: NIL
```

The STA instruction is now executed.

```
● (ApplyInstSD (SD (-- .MEM•(.PC);13,10≠0 $) $))

The pattern matcher returned:
pred: (SD (pre: .UPC=top .MEM•(.PC);13,10≠0)
          (mod: Q)
          (env:)
          (post: .OP=.MEM•(.PC);13,10 .UPC=addr .I=.MEM•(.PC)•15
                 (if .MEM•(.PC)•9=0
                     then .M=.MEM•(.PC);8
                   else .M=.MEM•(.PC);8+(LOGAND .PC 377000Q)))
          (vars:))
support: NIL

● (ApplyInstSD (SD (-- .I=0 $) $))

The pattern matcher returned:
```

```
pred: (SD (pre: .UPC=addr .I=0)
          (mod: UPC)
          (env:)
          (post: .UPC=action)
          (vars:))
support: NIL

• (Makeindexof 105110 .M=$)

The pattern matcher returned:
pred: (.M=105110)
support: ((M))

• (ApplyInstSD (SD (-- .OP=4 $) $)
               (m GFREET/MEM pc 105210))

The pattern matcher returned:
pred: (SD (pre: .UPC=action .OP=4 .M=m .PC=pc)
          (mod: Q PC MEM•m)
          (env:)
          (post: .UPC=top .PC=(pc+1);13 .MEM•m=.A)
          (vars: m pc))
support: NIL
```

The IRS instruction to increase the return address by 1 is now executed.

```
• (ApplyInstSD (SD (-- .MEM•(.PC);13,10≠0 $) $))

The pattern matcher returned:
pred: (SD (pre: .UPC=top .MEM•(.PC);13,10≠0)
          (mod: Q)
          (env:)
          (post: .OP=.MEM•(.PC);13,10 .UPC=addr .I=.MEM•(.PC)•15
                 (if .MEM•(.PC)•9=0
                     then .M=.MEM•(.PC);8
                   else .M=.MEM•(.PC);8+(LOGAND .PC 377000Q)))
          (vars:))
support: NIL

• (ApplyInstSD (SD (-- .I=0 $) $))
```

```
The pattern matcher returned:
pred: (SD (pre: .UPC=addr .I=0)
          (mod: UPC)
          (env:)
          (post: .UPC=action)
          (vars:))
support: NIL
```

```
● (Makeindexof 105120 .M=$)
```

```
The pattern matcher returned:
pred: (.M=105120)
support: ((M))
```

```
● (ApplyInstSD (SD (-- .OP=10 $) $ (-- m pc v))
               (m GFREE/MEM pc 105220 v ra))
```

```
The pattern matcher returned:
pred: (SD (pre: .UPC=action .OP=10 .M=m .MEM∘m=v .PC=pc)
          (mod: Q PC MEM∘m)
          (env:)
          (post: .UPC=top .MEM∘m=(v+1);15 (if .MEM∘m=0
                                              then (.PC=pc+2);13
                                             else .PC=(pc+1);13))
          (vars: m pc v))
support: NIL
```

As with the incrementing of NFS, it is necessary to show that the result of incrementing GFREE is not zero. The next six commands establish that .PC=105230 and that .GFREE=ra+1.

```
● (ApplyInstSD (SD (-- (x is less than y)) $)
               (x ra y 377770))
```

```
The pattern matcher returned:
pred: (SD (pre: (x is less than y))
          (mod:)
          (env:)
          (post: (x+1 is less than y+1))
          (vars: x y))
support: NIL

● (Underlimit (ra+1);15 .GFREE=$)

The pattern matcher returned:
pred: (.GFREE=(ra+1);15)
support: ((GFREE MEM=105120))

● (ApplyInstSD (SD (-- (0 is less than x)) $)
                (x ra+1))

The pattern matcher returned:
pred: (SD (pre: (0 is less than x))
          (mod:)
          (env:)
          (post: x≠0)
          (vars: x))
support: NIL

● (Substitute .GFREE ra+1 ra+1≠0)

The pattern matcher returned:
pred: (ra+1≠0)
support: NIL

● (Substitute (ra+1);15 .GFREE .GFREE≠0)

The pattern matcher returned:
pred: (.GFREE≠0)
support: ((GFREE MEM=105120))

● (SimpleEval (if (ra+1);15=0
                  then $
                else $))

The pattern matcher returned:
pred: (if (ra+1);15=0
          then (.PC=105240);13
        else .PC=105230)
support: ((PC))
```

The JMP to exit GFREE is now ready for execution. Because the jump is indirect through GFREE, bit 15 of the value in GFREE, i.e. ra+1, must be shown to be zero.

```
● (ApplyInstSD (SD (-- .MEM∘(.PC);13,10≠0 $) $))

The pattern matcher returned:
pred: (SD (pre: .UPC=top .MEM∘(.PC);13,10≠0)
          (mod: Q)
          (env:)
          (post: .OP=.MEM∘(.PC);13,10 .UPC=addr .. .MEM∘(.PC)∘15
                 (if .MEM∘(.PC)∘9=0
                     then .M=.MEM∘(.PC);8
                   else .M=.MEM∘(.PC);8+(LOGAND .PC 377000Q)))
          (vars:))
support: NIL

● (Makeindexof 105120 .M=$)

The pattern matcher returned:
pred: (.M=105120)
support: ((M))

● (ApplyInstSD (SD (-- .I=1 .M=m) $)
               (m GFREE/MEM))

The pattern matcher returned:
pred: (SD (pre: .UPC=addr .I=1 .M=m)
          (mod: UPC I M)
          (env:)
          (post: .UPC=addr .M=.MEM∘m;13 .I=.MEM∘m∘15)
          (vars: m))
support: NIL

● (ApplyInstSD (SD (-- (x is less than y)) $)
               (x ra y 37777Q))

The pattern matcher returned:
pred: (SD (pre: (x is less than y))
          (mod:)
          (env:)
          (post: (x+1 is less than y+1))
          (vars: x y))
```

```
support: NIL

• (Overlimit (ra+1)∘15 .I=$)

The pattern matcher returned:
pred: (.I=(ra+1)∘15)
support: ((I))

• (ApplyInstSD (SD (-- .I=0 $) $))

The pattern matcher returned:
pred: (SD (pre: .UPC=addr .I=0)
          (mod: UPC)
          (env:)
          (post: .UPC=action)
          (vars:))
support: NIL

• (Underlimit (ra+1);13 .M=$)

The pattern matcher returned:
pred: (.M=(ra+1);13)
support: ((M))

• (ApplyInstSD (SD (-- .OP=1 $) $ (-- m))
                (m ra+1))

The patterr matcher returned:
pred: (SD (pre: .UPC=action .OP=1 .M=m)
          (mod: Q PC)
          (env:)
          (post: .UPC=top .PC=m)
          (vars: m))
support: NIL
```

Execution is complete at this point. Before the subproof can be closed, however, each of the clauses in the postcondition of the goal must be derived. These are straightforward. The ForSome command produces the existential generalization of the specific case proven by the system. This exactly matches the form required in the postcondition of the goal and subproof is closed.

```
● (ApplyInstSD (SD $ (-- (x is pointing to y)) &)
               (x A y x∘0∘0))

The pattern matcher returned:
pred: (SD (pre: .x=y/MEM)
          (mod:)
          (env:)
          (post: (x is pointing to y))
          (vars: x y))
support: NIL

● (ApplyInstSD (SD $ (-- (x is pointing to y)) &)
               (x GFREET y x∘0∘0))

The pattern matcher returned:
pred: (SD (pre: .x=y/MEM)
          (mod:)
          (env:)
          (post: (x is pointing to y))
          (vars: x y))
support: NIL

● (ApplyInstSD (SD $ (-- (x is a permutation of <x∘0>@x,1)) &)
               (x x))

The pattern matcher returned:
pred: (SD (pre: x=<x∘0>@x,1)
          (mod:)
          (env:)
          (post: (x is a permutation of <x∘0>@x,1))
          (vars: x))
support: NIL

● (ApplyInstSD (SD $ (-- (x is a bufferlist beginning at y)) &)
               (x x,1 y p))

The pattern matcher returned:
pred: (SD (pre: (Subsetp x BufferSpace)
                y≠0
                (y is less than 37777Q)
                (if (LENGTH x)=0
                    then y=ZERO/MEM
                else x∘0 is a packet buffer and y=x∘0∘0/MEM
                      and x,1 is a bufferlist beginning at .x∘0∘0))
         (mod:)
         (env:)
         (post: (x is a bufferlist beginning at y))
         (vars: x y))
support: NIL
```

```
● (Substitute .FREE p (x,1 is a bufferlist beginning at p))
```

```
The pattern matcher returned:
pred: (x,1 is a bufferlist beginning at p)
support: ((x,1))
```

```
● (ForSome (y z)
          (x is a permutation of <y>@z and y is a packet buffer
              and z is a bufferlist beginning at .FREE and .ZERO=0
              and A is pointing to y∘0 and GFREET is pointing to y∘0)
          (y x∘0 z x,1))
```

```
● (Close)
```

At this point, the superior context is restored and the program counter and other parts of the state description are reset to just after the first instruction. The only changes to the current context are the addition of two SDs, one which states how execution would proceed if the free buffer list were empty and one which states how execution would proceed if the free buffer list were not empty.

These two SDs are now combined into a single SD using the CombineCases command. The precondition of the resulting SD is just

```
(LENGTH x)=0 or (LENGTH x)≠0
```

This is a specific instance of a tautology, and that specific instance is then derived for use in applying this SD.

```
● (CombineCases (SD (-- (LENGTH x)=0 $) $)
                (SD (-- (LENGTH x)≠0 $) $))
```

```
The pattern matcher returned:
```

```
pred: (SD (pre: (LENGTH x)≠0)
          (mod: x FREE GFREET GFREE A PC Q ZERO NFS)
          (env: OMEGA)
          (post: .UPC=top (LENGTH x)≠0 .PC=ra+1 .NFS=nfs+1
                (FS (y z)
                    (x is a permutation of <y>@z
                        and y is a packet buffer
                        and z is a bufferlist beginning at .FREE
                        and .ZERO=0 and A is pointing to y∘0
                        and GFREET is pointing to y∘0)))
          (vars:))
support: ((OMEGA))

pred: (SD (pre: (LENGTH x)=0)
          (mod: x FREE GFREET GFREE A PC Q ZERO)
          (env: OMEGA)
          (post: .UPC=top (LENGTH x)=0 .PC=ra
                (x is a bufferlist beginning at .FREE)
                .ZERO=0 .NFS=nfs)
          (vars:))
support: ((OMEGA))

● (ApplyInstSD (SD $ (-- (a=0 or a≠0)) &)
               (a (LENGTH x)))

The pattern matcher returned:
pred: (SD (pre:)
          (mod:)
          (env:)
          (post: (a=0 or a≠0))
          (vars: a))
support: NIL

● (ApplyInstSD (SD $))

The pattern matcher returned:
pred: (SD (pre: ((LENGTH x)=0 or (LENGTH x)≠0))
          (mod: x FREE GFREET GFREE A PC Q ZERO NFS)
          (env: OMEGA)
          (post: (.UPC=top and (LENGTH x)=0 and .PC=ra
                    and x is a bufferlist beginning at .FREE
                    and .ZERO=0 and .NFS=nfs or .UPC=top
                    and (LENGTH x)≠0
```

```
                      and .PC=ra+1 and .NFS=nfs+1
                      and (FS (y z)
                              (x is a permutation of <y>@z
                                   and y is a packet buffer
                                   and z is a bufferlist beginning at
                                       .FREE
                                   and .ZERO=0 and A is pointing to y∘0
                                   and GFREET is pointing to y∘0]
           (vars:))
support: ((OMEGA))
```

Execution of GFREE is now complete and the proofs corresponding to both the inner and outer SDs in the specification of GFREE can now be closed without further work.

- (Close)
- (Close)

6. Conclusions and Future Directions

Beyond the particular formulations presented here, there are number of concepts that have emerged in my mind as basic perceptions about program verification.

The first perception is that the way to specify the behavior of a (sequential) program is in terms of transitions between pairs of states. Using transitions to specify behavior provides a uniform mechanism for treating both known facts and desired intentions. The action of the individual instructions can be characterized in terms of state transitions, and these specifications can serve the role of axioms for a proof. The intended action of the whole program can also be formulated as a (set of) transition(s) to be proven. State deltas, of course, serve just this purpose.

Perception two is proofs should be structured. By a "structured" proof, I mean the following idea. A proof of correctness of a program consists of taking the SDs for the individual instructions and combining them into SDs which cover execution of sequences of instructions. This composition process is continued until the SDs which describe the intended behavior of the whole system are proven. These proofs should fall into three basic patterns, sequential proofs, proofs by cases, and inductive proofs.

> In a sequential proof, there are two SDs. The postcondition of the first leaves the machine in a state that satisfies the precondition of the second. These two SDs yield a new SD formed from the precondition of the first and the postcondition of the second. This pattern of reasoning applies directly to straightline code.
>
> In a proof by cases, the code under consideration contains conditional execution sequences or case statements. The preconditions of the given SDs represent alternative possibilities.[27] The SDs are combined by taking the

[27] These preconditions will usually be disjoint from each other, but this is not a requirement. The most common case for the preconditions of two SDs to overlap is for one of them to specify a short segment of computation and the other to specify a longer segment. Each SDs that was derived in the last chapter has a precondition that is strictly stronger than one of the input SDs. "Strictly stronger" means that it logically implies the precondition of the input SD. The fact that an interpretation already exists for sets of SDs which have overlapping preconditions means that any attempt to extend the use of SDs to model concurrent systems will need to use some mechanism other than sets of SDs which apply to the same states.

disjunction of the preconditions as the new precondition and the disjunction of the postconditions as the new postcondition.

In an inductive proof, the goal is to prove that the operation of a loop is correct. One SD covers the operation of the body of the loop, and the other covers the test which determines whether the loop is to be repeated or exited. The resulting SD covers the complete operation of the loop and (usually) is written in terms of some parameter that governs the number of times the loop is traversed. The induction proof has two parts. The first part proves that the operation of the loop is correct for the minimal value of the parameter. The second part assumes that the operation of the loop is correct for all values of the parameter below the value being considered and shows that the operation of the loop changes the state into a similar one with a smaller value of the parameter. The parameter must be chosen from a well-ordered set to give substance to the notions of "minimal" and "smaller" and to make use of the properties of mathematical induction.

Perception three is that the form of the proof and the form of the program need not bear strong resemblance. Efficient use of space or time frequently demands that the syntactic structure of the program be quite different from the conceptual structure. Some writers advocate that the structure of the program should exactly reflect the structure of the proof. When faced with the limitations in programming languages which lead programmers to perturb the organization of a program to achieve desired performance goals, these writers argue that the programming language is deficient and should be changed. My own position is that better programming languages are, of course, desirable, but no compiler can be smart enough to generate optimal code from the most readable statement of the algorithm. As a consequence, I believe proofs should not be bound intimately with the structure of the programming language.

Perception four is that it is reasonable to ask the programmer to supply the proof along with the program. Programming is a constructive process, and the person most knowledgeable about the interaction of the parts of the program is the person who put it together in the first place. It is sometimes argued that proving a program correct is much harder than writing and debugging the program to make it correct in the first place. This seems like pure nonsense to me. In order for the programmer to write and debug the program in the first place, he must have had an understanding of why the program would accomplish its goals. To be sure, his understanding is informal and not written out in terms acceptable to a program verification system. Consequently, there is some distance to be covered when we ask that formal proofs be carried out when a program is written. However, this distance is primarily one of building a language for the programmer to communicate his understanding of what he must already know.

Perception five is that it is at least as important to prove that a program actually progresses to an expected point as it is to prove that the result is correct when it gets there. This concept is usually labeled *termination*, but I think it is more appropriate to think in terms of *progress*. Many programs, e.g. operating systems, don't really terminate. They do, however, progress from one known state to the next, and the proof that each of these states is actually reached is of great concern. The idea of a set of "interesting" states instead of just an initial state and a final state was implied in the first perception by the use of a set of transitions instead of just a single transition. Proving that programs progress as intended comes for free in the proof patterns outlined in perception two. This is in marked contrast to the usual separation of proofs of correctness from proofs of termination when Floyd's method is used.

Perception six relates to the content of the state descriptions that make up the pre- and postconditions of a SD. These should be as general as possible. Facts which may be true in both the pre- and postcondition but which are not necessary to the correct operation of the code covered by the SD should not be included. For example, if an initial value is stored into a cell early in a program and the value is not used until much later, the SDs that cover the loops, subroutines and sequential segments in between these points should not have to mention the variable at all, much less retain its value.

Perception seven is that no state information should be hidden. Quite a bit of trouble is caused by pretending that machines have "automatic" mechanisms for invoking subroutines, iterating through loops, or remembering where the next data item is located. I have never found any way to represent these activities without keeping in mind that there was some part of the machine devoted to keeping track of the state.

Perception eight is that there must be a separate way to refer to the name of a place and to the current contents of that place. Arguments about indirect addressing, list processing, program modification and other matters depend upon this distinction very strongly.

Beyond these perceptions there are also a number of concrete ideas which point the way for improvement in both the theory and the implementation.

6.1 Theoretical issues

The most significant hole in the present theory is the absence of any mechanism to handle concurrent processes. The present formulation of state deltas is strongly dependent on the assumption that only one active agent is modifying the state of the system, and that it is sufficient to know what changes that agent is making to know all the changes that can possibly be made. At any given point in time, only a single "future" is possible. If more than one state delta happens to be applicable, it is understood that one of them carries the computation farther than the other.

To extend the present theory to handle concurrent systems, several modeling questions have to be answered. Are the concurrent systems to be composed of a small or large number of processors? If the number is small, we can imagine that each of the processors can have its own set of state deltas. On the other hand, if the system consists of a large number of identical or near identical processors, then some scheme is necessary to parameterize the SDs and perhaps parameterize the information in the contexts in the proof system. In either case, synchronization of the concurrent systems must be designed properly.

How do the processors connect to each other and what form is their communication? If the communication is restricted to signaling with semaphores and highly disciplined use of shared places, then some of the more modern formalisms may be applicable. Brinch-Hansen, Dijkstra, Estrin, Hoare and many others advocate that systems should be built with very restrictive rules governing the form of interprocess communication that is provided. For example, Estrin and his coworkers advocate separate explicit mechanisms governing control and associated data flow between modular subsystems. Existing systems, and many, many systems yet to be built, however, use simple shared places with no built-in restrictions on the access to these shared places by the processors. Usually these systems do adhere to a set of rules governing the communication, but the rules are not inherent in the structure. One of the more important facts to prove about these kinds of systems is that they compute the same result no matter how fast the processors run. If the processors do have a discipline which guarantees this fact, then the proof of uniqueness of the result will make use of the conventions followed by the several processors.

Considering the case of a small number of processors which are interconnected by shared places, we can see some of the changes in the present theory that will be necessary. State deltas now contain two lists of places, one which shows what set of places may be modified during the computation and one which shows which set must not have changed since the SD was proven. For concurrent systems with shared places, it seems necessary to add a similar list to show which places are referenced during the computation. If two processors are operating and one of them is modifying a place that the other is either modifying or reading, the result may be unpredictable. If the only concurrent use of shared places is that more than one process may be reading from the same location, then no conflict arises and the two operations may proceed in any order.

When more than one result is possible because of the order of execution, some scheme is necessary to examine all possible outcomes. The usual problem with this approach is that the number of cases grows exponentially and becomes intractable. The only hope for success is to lump the cases into large classes that behave similarly. Showing that all cases lumped into the same class do have the same effect would need to be proven, as well as proving that each of the classes behaves appropriately. Two ideas seem essential for this approach to succeed. First, some notion of "indivisible action" needs to be included in the formalism. As presently formulated, state deltas have no "grain size". Some set of SDs is introduced into the proof system from the description of the machine, but nothing prohibits the introduction of other SDs which cover smaller units of computation. In order to know what all the possible orders are among a set of competing processors, some notion of "grain size" is crucial.

Induction is the idea that seems necessary here. Each lumping of a set of possible execution sequences into a single class is probably based on an argument that the longer execution sequences reduce to a previously considered case after the first one or two steps have been taken. As we saw with sequential operation, the ability to apply normal induction rules to state deltas is extremely powerful and I would hope that a similar mechanism would emerge in the extension of the theory for concurrent operation.

The present formulation of the modification list raises an issue. At present, the semantics of a state delta say that values stored in locations not mentioned in the modification list are not changed during the course of the computation. This is considerably stronger than saying that the values of unmentioned places are the same after the computation as they are at the beginning. For sequential operation, the results are the same, but for concurrent operation these two formulations would have very different consequences. The present formulation was intended to lay the foundation for the concurrent case, so the stronger statement is necessary. There are instances, however, where the actual implementation of a program involves modification of a large number of places, but the values are restored when the computation terminates Garbage collection algorithms and operating system paging routines are two such examples. In our present formulation, we would have to include all of the places that are modified in the modification list, even though the values are restored. More importantly, we would no longer benefit from the implicit preservation of the state information. All predicates related to any of the modified places would have to be stated and reproven to hold. This requirement is completely contrary to our philosophy of representing just what the program knows and leaving the representation of the state information to the next higher level. Moreover, the mechanisms we have designed are strongly dependent on the fact that values are not touched. To extend the formalism to permit values to be changed but restored would also require a substantial change in the proof mechanisms.

It is not completely clear to me how all of these issues should be resolved, but it may be necessary to provide both kinds of state preservation. Perhaps a "restoration list" will have to be added to SDs, with the understanding that places listed on the restoration list may have had their values changed but that their values are restored at the end of the computation represented by the SD. Some experimentation is necessary.[28]

6.2 Engineering issues

No matter how the theory is extended to handle concurrency and multiple processors, any system will have most of its activity modeled as a sequential process. The present proof system is extremely clumsy and substantial work is needed to bring even the present theory into active use.

One of the more important weaknesses in the present proof system is the lack of mechanisms to represent arrays. Some means is needed to enter an array name into the place system and declare that it has, say, 16,384 elements. As the system stands now, this would have to be entered as a family in the place system with 16,384 daughters. Since each place node requires about 8 Interlisp cells, the overhead for this representation gets out of hand. Not all of this overhead can be avoided, however. Some of the cells in the memory array, for example, need to be represented explicitly because predicates are supported by the contents of those cells and not others. Most of the cells in memory, however, either contain pure code or are part of Bufferspace. Individual representation of these places within the place graph is unnecessary.

[28] Simulation is another area where the distinction between restoration and non-modification can be important. If an abstract machine is represented as having a set of disjoint locations and a set of high-level SDs that define its behavior, then it is not possible to implement this abstract machine with a low level machine that modifies but restores the values of the simulated disjoint places. As a consequence, some reasonable implementations of abstract machines are prohibited. There is one payoff: If machine C is proven to be a legal implementation of machine A, then an oscilloscopic probe may be place on the implementation of an abstract place and the probe will be active only when the abstract SDs predict change is possible. If we intend to marry the notions of simulation and interconnection of processors, we will want our formalism to guarantee that the substitution of a concrete implementation in place of an abstract definition will not change the operation of the total system.

Most of the predicates related to the state of the machine are of the form .X=v, where X is a place and v is a symbolic value that is not dependent upon the current state. Under these circumstances, it seems reasonable to streamline the representation of these predicates by allocating a dedicated cell for each place to hold its current value. Such a scheme comes much closer to the classical symbolic execution model and should have significant impact on the efficiency of the system.

A third area for improvement in representation is the SDs. The Honeywell 316 was modeled as having a microprogram counter. The role of the microprogram counter is to provide a method for distinguishing the substates within the execution of an instruction. In the simple model used in the main example, three substates are needed. In the translation of the full 316 description from ISPS, a few hundred substates are needed! Similarly, the number of distinct SDs needed for a machine description is fairly large: 9 for the simple machine and 310 for the full machine. In the specification of GFREE, the notion of a compound SD was introduced. The postcondition of a SD contains another SD. This idea can also be used in the description of the hardware. Instead of inventing a name for the intermediate values of the microprogram counter, it is possible to tie the SDs to its value and leave the value unspecified. Thus the only facts that need be known about the value of the microprogram counter are which SDs are valid. The following compound SDs could have been used to represent the simple version of the 316.

```
(Open (pre: (SD (pre: 316pre)
                (mod:)
                (env:)
                (post: (SD (pre: .PC=pc .MEM•(.PC)=140040Q)
                           (mod: Q PC A)
                           (env: UPC)
                           (post: 316pre .PC=(pc+1);13 .A=0)
                           (vars: pc))
                       (SD (pre: .PC=pc .MEM•(.PC)=101040Q)
                           (mod: Q PC)
                           (env: UPC)
                           (post: 316pre (if .A=0
                                              then .PC=(pc+1);13
                                            else .PC=(pc+2);13))
                           (vars: pc))
                       (SD (pre: .MEM•(.PC);13,10≠0)
                           (mod: Q addrctrl)
                           (env: UPC)
                           (post: addrpre (SD (pre: addrpost)
                                              (mod: UPC)
                                              (env: UPC addrctrl)
```

```
                         (post: 316pre)
                         (vars:))
                .OP=.MEM∘(.PC);13,10 .M;8=.MEM∘(.PC)
                ;8 (if .MEM∘(.PC)∘9=0
                     then .M;13.9=0
                   else .M;13,9=.PC;13,9)
                .I=.MEM∘(.PC)∘15)
           (vars:)))
   (vars:))
(SD (pre: addrpre)
   (mod:)
   (env:)
   (post: (SD (pre: .I=0)
              (mod: UPC actionctrl)
              (env: UPC)
              (post: actionpre (SD (pre: actionpost)
                                   (mod: UPC)
                                   (env: UPC actionctrl)
                                   (post: addrpost)
                                   (vars:)))
              (vars:))
           (SD (pre: addrpre .I=1 .M=m)
              (mod: UPC I M)
              (env: UPC)
              (post: addrpre .M=.MEM∘m;13 .I=.MEM∘m∘15)
              (vars: m)))
   (vars:))
(SD (pre: actionpre)
   (mod:)
   (env: UPC)
   (post: (SD (pre: .OP=1 .M=m)
              (mod: Q PC)
              (env: UPC)
              (post: actionpost .PC=m)
              (vars: m))
           (SD (pre: .OP=4 .M=m .PC=pc)
              (mod: Q PC MEM∘m)
              (env: UPC)
              (post: actionpost .PC=(pc+1);13 .MEM∘m=.A)
              (vars: m pc))
           (SD (pre: .OP=10 .M=m .MEM∘m=v .PC=pc)
              (mod: Q PC MEM∘m)
              (env: UPC)
```

```
                           (post: actionpost .MEM◦m=(v+1);15
                                  (if .MEM◦m=0
                                      then (.PC=pc+2);13
                                    else .PC=(pc+1);13))
                           (vars: m pc v))
                       (SD (pre: .OP=11 .PC=pc .A=a .M=m .MEM◦m=b)
                           (mod: Q PC A MEM◦m)
                           (env: UPC)
                           (post: actionpost .PC=(pc+1);13 .A=b
                                  .MEM◦m=a)
                           (vars: pc m a b)))
                (vars:))
            (Covering OMEGA
                      <addrctrl actionctrl UPC PC MEM M I A C OP>)
            (Covering MEM
                      <MEM◦277Q MEM◦500Q MEM◦105110 MEM◦10512Q
                        MEM◦10513Q MEM◦10514Q MEM◦10515Q MEM◦10516Q
                        MEM◦10517Q MEM◦10520Q MEM◦10521Q MEM◦10522Q
                        MEM◦10523Q BufferSpace ZERO>))
      (mod: OMEGA)
      (env:)
      (post:)
      (vars:))
(NewDecomposition (Covering OMEGA
                            <addrctrl actionctrl UPC PC MEM M I A C OP>)
                  )
(NewDecomposition (Covering MEM
                            <MEM◦277Q MEM◦500Q MEM◦10511Q MEM◦10512Q
                              MEM◦10513Q MEM◦10514Q MEM◦10515Q
                              MEM◦10516Q MEM◦10517Q MEM◦10520Q
                              MEM◦10521Q MEM◦10522Q MEM◦10523Q
                              BufferSpace ZERO>))
(NewComposition (Covering Q <UPC M I OP>))
```

Only three SDs exist at the top level. These stay in existence permanently. When one of them is activated, it may bring one or more others into existence. However, application of any of these new SDs causes all of the new SDs to be deleted from the current context, although others may be added as the result of the application. The number of compound SDs that need to be available permanently seems to be based on the number of loops in the machine description. for the full 316, three loops exist -- the top level, the shift cycle and the indirect addressing cycle -- so the 310 SDs could be reduced to 3 permanent SDs and a small handful of active SDs at each step.

Reduction of the number of SDs makes it possible to build a reasonable proposer to select which SD to apply next. Brute force testing of all of the preconditions is not even ruled out, but more sophisticated schemes may be possible. Any improvement in the proposer will have a first-order effect on the size of the proof.

The use of the pattern matcher to select predicates is extraordinarily expensive. The pattern matcher we are using compiles its patterns when they are first encountered because it expects to use them repetitively. We use our patterns exactly once. A large portion of the execution time in the present system is spent converting these patterns to executable code. The space consumed by the translations is freed up after it is used, and a large number of garbage collections are needed during the course of the proof to recover this space. Finally, the patterns are used to search the whole list of predicates accessible in the current context. As contexts become larger and larger, this strategy becomes infeasible. Some means to refer to predicates needs to be found which does not involve searching the whole context and which does not consume and discard a lot of space. Predicates are already cross-referenced according to their support, and this provides at least one useful way of reaching predicates quickly. Many predicates have no support, however, and perhaps an additional cross-referencing scheme based on free variables will be useful. SDs may need to be cross-referenced according to the components of their preconditions. If we view the SDs as a set of productions, cross-referencing them according to their preconditions provides the basis for a powerful control mechanism for selecting and applying the right SD with little cost.[29]

6.3 Prognosis

If we look forward to a time when verification is an accepted practice and the programmer submits his program to a verifier with the same regularity that he now submits it to a compiler or assembler, three perceptions emerge. First, the verification must be completely automatic and deterministic. Second, the verifier must process between ten and one hundred statements per second.[30] Third, the proof may not be much longer than the program but need not be much shorter.

[29] Randall Davis and Jonathan King, "An Overview of Production Systems," Computer Science Department, Stanford University, California, STAN-CS-75-524, October 1975.

[30] I am indebted to Mac McKinley for posing the question of how long it would take to do a full verification.

Requiring the verifier to operate automatically is quite distinct from the issue of development of the proof interactively. I believe that development of the proof is very likely to be an interactive process and be comparable in style to the use of an editor. Once complete, however, the program and its proof will be assembled or compiled to executable code and at the same time verified for correctness. Programmers generally expect compilation errors only if they mistype. Compilers which are very poorly documented, in a state of transition or syntactically arcane provide the alternate surprise that the program may look correct even upon close inspection but may fail to compile because the compiler doesn't work as expected. Such compilers are held in low esteem and in practice do not survive long. I expect verification systems to live within the same constraints, and thus programmers will expect to be able to write correct proofs with a high degree of confidence and have these proofs be accepted with the same success that source code passes the syntax checker. Wegbreit's recent work looks very promising along this line.[31]

Given that we accept the need for the verifier to be automatic, we can further look at the effect of how the speed of the verification system will affect its use. In our present environment assembly of a large program (e.g. the TENEX operating system) requires approximately an hour to assemble a quarter million instructions. Full assemblies are done infrequently but probably as often as once a day as a new version of the system nears completion or is assembled with local parameters for particularization to a site. Smaller programs, say ten thousand instructions, require a couple of minutes and a user will reassemble at almost every convenient juncture. If we intend that verification of a program fit into this mold and thus bring a milieu in which programs are checked for consistency with their specifications as naturally as they are checked for correct syntax, then the verification system will have to perform at comparable speeds, say close to 100 instructions per second. Performance in the range of 10 instructions per second means that a user will reverify a 10,000 instruction program using an overnight batch service and will tolerate more frequent verification only for programs up to 1000 words. These numbers are comparable to the performance levels of poor compilers in the 1960's or earlier and may well lie within the tolerable range for consistent use. However, the increased cost and delay will need to be offset by demonstrated payoff in the location of bugs or by management edict.

Performance in the range of one statement per second is likely to inhibit wholesale adoption of verification as a production tool and restrict its use to experimental programming groups and selected critical system development efforts.

[31] Ben Wegbreit, "Constructive Methods in Program Verification," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 3, May 1977, pp. 193-209.

I have not begun to perform detailed measurements, nor is it entirely meaningful to do so at this time. At present the proofs are so laborious that even an infinitely fast verifier would not be attractive enough to overcome the excessive labor required to prepare the proof. These caveats aside, I believe the current system verifies between one-tenth and one statement per second and is thus acceptable as an experimental vehicle but unacceptable as a production tool.

With respect to the size of the proof, we can use the same kind of "impedance match" argument that the proof must not be too much longer than the program. Long proofs require more labor by the programmer and will be avoided. Proofs that do not materially change the coding time are thus required for regular use. Again Wegbreit's recent work suggests this goal is attainable without substantial difficulty. Our present system is well short of the mark. Our first proof was nine pages long for a nine instruction program and is further embarrassed by the presence of unchecked user-supplied ellisions in the reasoning chain. A ratio of 50 to one is laughably unacceptable, but is not much cause for worry; ideas for automating the proof system and compressing the proof are flowing so rapidly that the only role of the present statistic is to set the stage for spectacular claims of improvement in the future.

Bibliography

Barbacci, Mario R., Gary E. Barnes, Roderic G. Cattell and Daniel P. Siewiorek, "The Symbolic Manipulation of Computer Descriptions: ISPS Primer," Departments of Computer Science and Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pennsylvania, August 14, 1977.

Bell, C. Gordon and Allen Newell, *Computer Structures: Readings and Examples*, McGraw-Hill Book Company, New York, 1971.

Birman, A. and W. H. Joyner Jr., "A Problem-Reduction Approach to Proving Simulation Between Programs," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 2, pp. 87-96, June 1976.

Burstall, R. M., "Program Proving as Hand Simulation with a Little Induction," *Information Processing 1974*, *Proceedings of the IFIP Congress*, North-Holland Publishing Company, Amsterdam, pp. 308-312.

Burstall, R. M., "Some Techniques for Proving Correctness of Programs which Alter Data Structures," *Machine Intelligence 7*, Edinburgh University Press, Edinburgh, Scotland, 1972, pp. 23-50.

Chirica, L. M., "Contributions to Compiler Correctness," Computer Science Department, School of Engineering and Applied Science, University of California, Los Angeles, UCLA-ENG-7697, October 1976.

Davis, Randall and Jonathan King, "An Oveview of Production Systems," Computer Science Department, Stanford University, California, STAN-CS-75-524, October 1975.

Fikes, R. E. and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence*, Vol. 2, Nos. 3 and 4, pp. 189-208, 1971.

Fikes, R. E., P. E. Hart and N. J. Nilsson, "Learning and Executing Generalized Robot Plans," *Artificial Intelligence*, Vol. 3, pp. 251-288, 1972.

Floyd, R. W., "Assigning Meanings to Programs," *Mathematical Aspects of Computer Science*, Vol. XIX, Proceedings of Symposia in Applied Mathematics, American Mathematical Society, Providence, Rhode Island, 1967, pp. 19-32.

Good, Donald I., Ralph L. London, and W. W. Bledsoe, "An Interactive Program Verification System," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1, pp. 59-67, March 1975.

Heart, F.E., et al., "The Interface Message Processor for the ARPA Computer Network," *Proceedings of the AFIPS Spring Joint Computer Conference*, Vol. 36, American Federation of Information Processing Societies, Montvale, New Jersey, 1970, pp. 551-567.

Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, Vol. 12, No. 2, pp. 576-583, October 1969.

Hoare, C.A.R. and P.E. Lauer, "Consistent and Complementary Formal Theories of the Semantics of Programming Languages," Computing Laboratory, Claremont Tower, University of Newcastle upon Tyne, England, Technical Report 44, April 1973.

Hoare, C.A.R., "Parallel Programming: An Axiomatic Approach," *Computer Languages*, Vol. 1, No. 2, pp. 151-160, June 1975.

Hoare, C.A.R., "Towards a Theory of Parallel Programming," *Operating Systems Techniques*, edited by C.A.R. Hoare, Academic Press, pp. 61-71, 1972.

Honeywell, Inc., *Programmers Reference Manual: DDP-516 General Purpose Computer*, Framingham, Massachusetts, August 1968.

Igarashi, Shigeru, Ralph L. London, and David C. Luckham, "Automatic Program Verification I: A Logical Basis and Its Implementation," *Acta Informatica*, Vol. 4, No. 2, pp. 145-182, 1975.

Kalish, Donald and Richard Montague, *LOGIC: Techniques of Formal Reasoning*, Harcourt, Brace and World, Inc., New York, 1964.

King, James C. "Symbolic Execution and Program Testing," *IBM Research*, RC 5082, October 18, 1974.

London, Ralph L., "Perspectives on Program Verification," *Current Trends in Programming Methodology*, R. T. Yeh (ed.), Vol II, Prentice-Hall Book Company, Inc., 1977. (To appear.)

Luckham, David C. and Norihisa Suzuki, *Automatic Program Verification IV: Proof of Termination within a Weak Logic of Programs*, Computer Science Department, Stanford University, California STAN-CS-75-522, October 1975.

Manna, Zohar and Richard Waldinger, "Is 'Sometime' Sometimes Better than 'Always'? Intermittent Assertions in Proving Program Correctness," *Communications of the ACM*, (to appear).

McCarthy, John, "Situations, Actions and Causal Laws," *Semantic Information Processing*, edited by Marvin Minsky, MIT Press, Cambridge, Massachusetts, 1968, pp. 410-417.

McCarthy, John, and P.J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," *Machine Intelligence 4*, edited by B. Meltzer and D. Michie, American Elsevier Publishing Co., Inc., New York, 1969, pp. 463-502.

McDermott, J., et al., "The Efficiency of Certain Production System Implementations," Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, September 1976.

McDermott, J. and C. Forgy, "Production System Conflict Resolution Stategies," Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, December 1976.

Patterson, D., "Verification of Microprograms," Computer Science Department, School of Engineering and Applied Science, University of California, Los Angeles, UCLA-ENG-7707, January 1977.

Poupon, Jacques and Ben Wegbreit, "Covering Functions," Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, September 1972.

A Research Program in Computer Technology: Annual Technical Report, May 1974-June 1975, USC/Information Sciences Institute, ISI/SR-75-3, September 1975.

A Research Program in Computer Technology: Annual Technical Report, July 1975-June 1976, USC/Information Sciences Institute, ISI/SR-76-6, July 1976.

Roberts, L. G. and B. D. Wessler, "Computer Network Development to Achieve Resource Sharing," *Proceedings of the AFIPS Spring Joint Computer Conference*, Vol. 36, American Federation of Information Processing Societies, Montvale, New Jersey, 1970, pp. 543-549.

Sites, Richard L., *Proving that Computer Programs Terminate Cleanly*, Computer Science Department, Stanford University, California, STAN-CS-74-418, May 1974.

Sites, Richard L., *Some Thoughts on Proving Clean Termination of Programs*, Computer Science Department, Stanford University, California, STAN-CS-74-417, May 1974.

Suzuki, Norihisa, *Automatic Verification of Programs with Complex Data Structures*, Computer Science Department, Stanford University, California STAN-CS-76-552, February 1976.

Teitelman, Warren, *Interlisp Reference Manual*, Xerox Palo Alto Research Center, Palo Alto, California, 1975.

Wegbreit, B., "Constructive Methods in Program Verification". *IEEE Transactions on Software Engineering*, Vol. SE-3, pp. 193-209, No. 3, May 1977.